Einführung in C/C++

Wulf Alex

2008

Karlsruhe

Ausgabedatum: 18. November 2008.

Email: alex-weingarten@t-online.de

Dies ist ein Skriptum. Es ist unvollständig und enthält Fehler.

Geschrieben mit dem Texteditor `vi`, formatiert mit LaTeX unter Debian GNU/Linux.

Die Skripten liegen unter folgenden URLs zum Herunterladen bereit:

`http://www.alex-weingarten.de/skripten/`
`http://www.abklex.de/skripten/`

Besuchen Sie auch die Seiten zu meinen Büchern:

`http://www.alex-weingarten.de/debian/`
`http://www.abklex.de/debian/`

Von dem Skriptum gibt es neben der Normalausgabe eine Ausgabe in kleinerer Schrift (9 Punkte), in großer Schrift (14 Punkte) sowie eine Textausgabe für Leseprogramme (Screenreader).

There is an old system called UNIX,
suspected by many to do nix,
but in fact it does more
than all systems before,
and comprises astonishing uniques.

Vorwort

Die Skripten richten sich an Leser mit wenigen Vorkenntnissen in der Elektronischen Datenverarbeitung; sie sollen – wie FRITZ REUTERS *Urgeschicht von Meckelnborg* – ok för Schaulkinner tau bruken sin. Für die wissenschaftliche Welt zitiere ich aus dem Vorwort zu einem Buch des Mathematikers RICHARD COURANT: *Das Buch wendet sich an einen weiten Kreis: an Schüler und Lehrer, an Anfänger und Gelehrte, an Philosophen und Ingenieure*. Das Lernziel ist eine Vertrautheit mit Betriebssystemen der Gattung UNIX einschließlich Linux, der Programmiersprache C/C++ und dem weltumspannenden Internet, die so weit reicht, dass der Leser mit der Praxis beginnen und sich selbständig weiterbilden kann. Ausgelernt hat man nie.

Zusammen bildeten die Skripten die Grundlage für das Buch *UNIX. C und Internet*, im Jahr 1999 in zweiter Auflage im Springer-Verlag erschienen (ISBN 3-540-65429-1). Das Buch ist vergriffen und wird auch nicht weiter gepflegt, da ich mich auf Debian GNU/Linux konzentriere. Meine Debian-Bücher (ISBN 3-540-43267-1 und 3-540-23786-0) sind ebenfalls bei Springer erschienen, aber nicht im Netz veröffentlicht. Die Skripten dagegen bleiben weiterhin im Netz verfügbar und werden bei Gelegenheit immer wieder überarbeitet.

Warum ein Linux/UNIX? Die Betriebssysteme der Gattung UNIX laufen auf einer Vielzahl von Computertypen. Unter den verbreiteten Betriebssystemen sind sie die ältesten und ausgereift. Die UNIXe haben sich lange ohne kommerzielle Einflüsse entwickelt und tun das teilweise heute noch, siehe Linux, FreeBSD, NetBSD, OpenBSD und andere. Programmierer, nicht das Marketing, haben die Ziele gesetzt. Die UNIXe haben von Anfang an gemischte Hardware und die Zusammenarbeit mehrerer Benutzer unterstützt. In Verbindung mit dem X Window System, einem netzfähigen Fenstersystem, sind die UNIXe unter den Betriebssystemen mittlerer Größe die leistungsfähigsten. Linux/UNIX-Rechner waren von Anbeginn im Internet dabei und haben seine Entwicklung bestimmt.

Warum C/C++? Die universelle Programmiersprache C mit ihrer mächtigen Erweiterung C++ ist – im Vergleich zu BASIC etwa – ziemlich einheitlich. Der Anfang ist leicht, an die Grenzen stoßen wenige Benutzer. Das Zusammenspiel zwischen C/C++-Programmen und Linux/UNIX funktioniert reibungslos.

Warum das Internet? Das Internet ist das größte Computernetz dieser Erde, ein Zusammenschluss vieler regionaler Netze. Ursprünglich auf Hochschulen und Behörden beschränkt, sind mittlerweile auch Industrie, Handel und Privatpersonen beteiligt. Unser berufliches Leben und zunehmend unser privates Dasein werden vom Internet berührt. Eine Email-Anschrift ist so wichtig geworden wie ein Telefonanschluss. Als Informationsquelle ist das Netz unentbehrlich.

Bei der Stoffauswahl habe ich mich von meiner Arbeit als Benutzer, Verwalter und Programmierer leiten lassen. Besonderer Wert wird auf die Erläuterung der zahlreichen Fachbegriffe gelegt, die dem Anfänger das Leben erschweren. Die typische Frage, vor der auch ich immer wieder stehe, lautet: *Was ist XYZ und wozu kann man es gebrauchen?* Hinsichtlich vieler Einzelheiten verweise ich auf die Referenz-Handbücher zu den Rechenanlagen und Programmiersprachen oder auf Monografien, um den Text nicht über die Maßen aufzublähen; er ist ein Kompromiss aus Breite und Tiefe. *Alles über UNIX, C und das Internet* ist kein Buch, sondern ein Bücherschrank.

An einigen Stellen gehe ich außer auf das Wie auch auf das Warum ein. Von Zeit zu Zeit sollte man den Blick weg von den Wellen auf das Meer richten, sonst erwirbt man nur kurzlebiges Wissen.

Man kann den Gebrauch eines Betriebssystems, einer Programmiersprache oder der

Netzdienste nicht allein aus Büchern erlernen – das ist wie beim Klavierspielen oder Kuchenbacken. Die Beispiele und Übungen wurden auf einer Hewlett-Packard 9000/712 unter HP-UX 10.20 und einem PC der Marke *Weingartener Katzenberg Auslese* unter Debian GNU/Linux entwickelt. Als Shell wurden Bourne-Abkömmlinge bevorzugt, als Compiler wurde neben dem von Hewlett-Packard der GNU gcc verwendet. Die vollständigen Quellen der Beispiele stehen im Netz.

Dem Text liegen eigene Erfahrungen aus fünf Jahrzehnten zugrunde. Seine Wurzeln gehen zurück auf eine *Erste Hilfe für Benutzer der Hewlett-Packard 9000 Modell 550 unter HP-UX*, im Jahr 1986 aus zwanzig Aktenordnern destilliert, die die Maschine begleiteten. Gegenwärtig verschiebt sich der Schwerpunkt in Richtung Debian GNU/Linux. Ich habe auch fremde Hilfe beansprucht und danke Kollegen in den Universitäten Karlsruhe und Lyon sowie Mitarbeitern der Firmen IBM und Hewlett-Packard für schriftliche Unterlagen und mündlichen Rat sowie zahlreichen Studenten für Anregungen und Diskussionen. Darüber hinaus habe ich fleißig das Internet angezapft und viele dort umlaufende Guides, Primers, HOWTOs, Tutorials und Sammlungen von Frequently Asked Questions (FAQs) verwendet.

Weingarten (Baden), 15. Februar 2006 — Wulf Alex

Übersicht

Inhalt

Abbildungen

Tabellen

Programme und andere Quellen

Zum Gebrauch

- Hervorhebungen im Text werden *kursiv* dargestellt.
- Titel von Veröffentlichungen oder Abschnitten, kurze Zitate oder wörtliche Rede werden im Text *kursiv* markiert.
- In Aussagen über Wörter werden diese *kursiv* abgesetzt.
- Stichwörter für einen Vortrag oder eine Vorlesung erscheinen **fett**.
- Namen von Personen stehen in KAPITÄLCHEN.
- Eingaben von der Tastatur und Ausgaben auf den Bildschirm werden in `Schreibmaschinenschrift` wiedergegeben.
- Hinsichtlich der deutschen Rechtschreibung befindet sich das Manuskript in einem Übergangsstadium.
- Hinter Linux/UNIX-Kommandos folgt manchmal in Klammern die Nummer der betroffenen Sektion des Referenz-Handbuchs, z. B. `vi(1)`. Diese Nummer samt Klammern ist beim Aufruf des Kommandos nicht einzugeben.
- Suchen Sie die englische oder französische Übersetzung eines deutschen Fachwortes, so finden Sie diese bei der erstmaligen Erläuterung des deutschen Wortes.
- Suchen Sie die deutsche Übersetzung eines englischen oder französischen Fachwortes, so finden Sie einen Verweis im Sach- und Namensverzeichnis.
- UNIX wird hier immer als die Gattung der aus dem bei AT&T um 1970 entwickelten Unix ähnlichen Betriebssysteme verstanden, nicht als geschützter Name eines bestimmten Produktes.
- Ich gebe möglichst genaue Hinweise auf weiterführende Dokumente im Netz. Der Leser sei sich aber bewußt, dass sich sowohl Inhalte wie Adressen (URLs) ändern. Bei Verweisen auf Webseiten (URLs) ist die Angabe des Protokolls `http://` weggelassen.
- Unter *Benutzer, Programmierer, Verwalter* usw. werden sowohl männliche wie weibliche Erscheinungsformen verstanden.
- Ich rede den Leser mit *Sie* an, obwohl unter Studenten und im Netz das *Du* üblich ist. Gegenwärtig erscheint mir diese Wahl passender.

Brevity is the soul of wit.
Shakespeare, Hamlet

1 Programmieren in C/C++

1.1 Grundbegriffe

1.1.1 Wozu Programmierkenntnisse?

Auch wer seine Brötchen nicht mit Programmieren verdient, sondern nur mit Computern arbeitet, braucht Grundkenntnisse im Programmieren, um die richtigen Vorstellungen von dem Geschehen hinter dem Bildschirm zu haben. Nicht jeder Maschinenbauer muss einen Kolbenmotor konstruieren können, aber er muss wissen, wie eine Kolbenmaschine aufgebaut ist und funktioniert, und er sollte mit einem Schraubenschlüssel umgehen können. Hinzu kommt, dass häufig kleinere Anpassungen oder Ergänzungen der Software erforderlich sind, für die man nicht immer einen Spezialisten hat.

Wenn es ernst wird, bezieht man seine Kenntnisse aus drei Quellen, nämlich aus:

- einem Lehrbuch wie diesem,
- einer Referenz zum Nachschlagen, siehe das Verzeichnis am Ende des Buches,
- einer Sammlung von *Frequently Asked Questions* aus dem Internet, hier zu den Themen C und C++.

Das Lehrbuch sorgt für den Überblick, die Referenz vermittelt Einzelheiten, die FAQ beantwortet die offen gebliebenen Fragen. FAQs zu C und C++ finden sich unter:

- `www.eskimo.com/ scs/c-faq/top.html`, die FAQ der Newsgruppe `comp.lang.c`,
- `www-info2.informatik.uni-wuerzburg.de/dclc-faq/`, die FAQ der Newsgruppe `de.comp.lang.c`,
- `www.faqs.org/faqs/C-faq/learn/`,
- `www.cerfnet.com/~mpcline/c++-faq-lite/`,
- `rtfm.mit.edu/pub/usenet-by-hierarchy/news/answers/C++-faq/` bzw. `C-faq/`.

Langfristig sind auch die genannen Newsgruppen eine Quelle der Erleuchtung, aber man sollte Grundkenntnisse erworben haben, ehe man dort mitmischt.

1.1.2 Warum braucht man Programmiersprachen?

Von einer Anweisung in einer höheren Programmiersprache bis zu den Nullen und Einsen im Befehlsregister des Prozessors ist ein weiter Weg. Wir wollen diesen Weg schrittweise an Hand eines kleinen, aber weltweit bekannten Programmes verfolgen. Das Programm schreibt den Gruß *Hallo, Welt!* auf den Bildschirm. Weitere Exemplare dieses Programmes in über 200 Programmiersprachen[1] finden sich bei der Louisiana Tech University unter:

`http://www.latech.edu/~acm/HelloWorld.shtml`

Als erstes das Programm, so wie es ein C-Programmierer schreibt:

[1]Unter `www.ionet.net/~timtroyr/funhouse/beer.html` liegt eine ähnliche Sammlung.

```
/* hallo.c, C-Programm */

#include <stdio.h>

int main()
{
printf("Hallo, Welt!\n");
return 0;
}
```

Das Aussehen des Textes wird durch den ANSI-C-Standard bestimmt, letzten Endes durch die Leute, die die Sprache C entwickelt haben. Diese Form des Programmes wird von geübten Programmierern verstanden und **Programmquelle** (source) genannt. Die Maschine kann nichts damit anfangen.

Damit das Programm von einer Maschine ausgeführt werden kann, muss es übersetzt werden. Hierzu wird ein weiteres Programm, ein Compiler, herangezogen. Im Fall von C läuft der Übersetzungsvorgang in mehreren Schritten ab, ohne dass der Benutzer etwas davon merkt. Wir verwenden hier den GNU-C-Compiler unter MS-DOS auf einem PC. Im ersten Schritt (Präprozessor) werden der für die Maschine unbedeutende Kommentar entfernt und die mit einem Doppelkreuz beginnenden Präprozessor-Anweisungen ausgeführt. Das Ergebnis sieht leicht gekürzt so aus:

```
# 1 "hallo.c"
# 1 "c:/djgpp/include/stdio.h" 1 3
# 1 "c:/djgpp/include/sys/djtypes.h" 1 3
# 12 "c:/djgpp/include/stdio.h" 2 3

typedef void *va_list;
typedef long unsigned int size_t;
typedef struct {
  int   _cnt;
  char *_ptr;
  char *_base;
  int   _bufsiz;
  int   _flag;
  int   _file;
  char *_name_to_remove;
} FILE;

extern FILE __dj_stdin, __dj_stdout, __dj_stderr;

void    clearerr(FILE *_stream);
int     fclose(FILE *_stream);
int     feof(FILE *_stream);
.
.
int     printf(const char *_format, ...);
.
.
int     vsprintf(char *_s, const char *_format, va_list _ap);

extern FILE __dj_stdprn, __dj_stdaux;

# 3 "hallo.c" 2

int main()
{
printf("Hallo, Welt!\n");
return 0;
}
```

Wir erkennen, dass der Präprozessor eine Reihe von Zeilen hinzugefügt hat. Im Prinzip könnte das auch der Programmierer machen, doch so erspart man sich viel routinemäßige Arbeit.

Im zweiten Schritt wird das C-Programm in ein Assembler-Programm übersetzt:

```
        .file   "hallo.c"
```

```
gcc2_compiled.:
___gnu_compiled_c:
.text
LC0:
        .ascii "Hallo, Welt!\12\0"
        .align 2
.globl _main
_main:
        pushl %ebp
        movl %esp,%ebp
        call ___main
        pushl $LC0
        call _printf
        addl $4,%esp
        xorl %eax,%eax
        jmp L1
        .align 2,0x90
L1:
        leave
        ret
```

Selbst diese, bereits schwerer verständliche Form könnte ein erfahrener Programmierer noch von Hand schreiben. Früher gab es nichts anderes. Im Großen und Ganzen entspricht eine Anweisung in C oder FORTRAN vier Assembler-Anweisungen. Die Assembler-Befehle werden zu einem wesentlichen Teil durch den Hersteller der CPU bestimmt, hier also durch Intel. Das Assembler-Programm ist an die Hardware und das Betriebssystem gebunden.

Nun folgt als dritter Schritt die Übersetzung des Assemblerprogramms in ein Maschinenprogramm, hier gekürzt und mit Hexadezimalzahlen anstelle der Nullen und Einsen wiedergegeben:

```
4c01 0300 66da 7d31 da00 0000 0d00 0000
0000 0401 2e74 6578 7400 0000 0000 0000
0000 0000 3000 0000 8c00 0000 bc00 0000
0000 0000 0300 0000 2000 0000 2e64 6174
6100 0000 3000 0000 3000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
4000 0000 2e62 7373 0000 0000 3000 0000
```

Diese Form ist für einen Menschen nicht mehr verständlich, stattdessen für die Maschine, weshalb sie als Maschinencode bezeichnet wird, gelegentlich auch als Objektcode. Ein Zurück-Übersetzen in Assemblercode ist nur eingeschränkt möglich[2].

Obige Form ist jedoch immer noch nicht ausführbar. Wir verwenden eine Standard-Funktion `printf()` zur Ausgabe auf den Bildschirm. Auch hinter dem Wörtchen `main` verbirgt sich einiges. Deren Code muß noch hinzugefügt werden, dann kann die Maschine loslegen mit der Begrüßung. Diesen letzten Schritt vollzieht der Linker. Der Anfang des ausführbaren Programmes `hallo.exe` sieht nicht besser aus als vorher, der Umfang des Programmfiles ist größer geworden:

```
4d5a 0000 0400 0000 2000 2700 ffff 0000
6007 0000 5400 0000 0d0a 7374 7562 2e68
2067 656e 6572 6174 6564 2066 726f 6d20
7374 7562 2e61 736d 2062 7920 646a 6173
6d2c 206f 6e20 5475 6520 4a61 6e20 3330
2032 333a 3433 3a35 3820 3139 3936 0d0a
5468 6520 5354 5542 2e45 5845 2073 7475
```

Das müsste ein Programmierer schreiben, gäbe es keine höheren Programmiersprachen. Als Kontrast dazu ein kurzes Beispiel einer problemangepassten, maschinenfernen Sprache (SQL). Die Aufgabe sei die Abfrage einer Datenbank, die ihre Daten in Form von Tabellen mit Spalten und Zeilen hält:

[2]Genauer: Das Rück-Übersetzen ist möglich, das Ergebnis jedoch praktisch unbrauchbar, wenn es um mehr als triviale Programme geht.

```
select nachname, vorname, telefon from mitarbeiter
where wohnort='Karlsruhe'
order by nachname, vorname;
```

Die Datenbank soll bitteschön ein Liste mit Nachnamen, Vornamen und Telefonnummer aus der Tabelle `mitarbeiter` herausziehen und dabei nur die Mitarbeiter berücksichtigen, deren Wohnort Karlsruhe ist. Die Liste soll an erster Stelle nach dem Nachnamen alphabetisch sortiert sein, bei gleichem Nachnamen nach dem Vornamen. Einfacher lässt sich eine Aufgabe kaum formulieren.

Eine Programmiersprache wird von zwei Seiten her entwickelt. Von oben, den zu programmierenden Aufgaben aus der realen Welt her, kommen die Anforderungen an die Sprache. Von unten, der Hardware (CPU) und dem Betriebssystem her kommen die Möglichkeiten zur Lösung der Aufgaben. Wir haben folgende Schichten:

- Aufgabe, Problem
- Lösungsweg, Algorithmus
- Programm in einer höheren (problemorientierten) Sprache
- Assemblerprogramm
- Maschinenprogramm
- Mikroprogramme (Firmware) im Computer
- Elektronik

Jede Schicht stellt ein Modell der nächsthöheren Schicht dar, wobei das, was sich in der Elektronik abspielt, hoffentlich noch etwas mit der ursprünglichen Aufgabe zu tun hat. Der Compilerbauer muss sowohl das Problem wie die Hardware samt Betriebssystem im Auge haben, wenn er beispielsweise einen C-Compiler für das Betriebssystem PC-DOS auf einem Intel-Prozessor schreibt. Wer sich näher für Compiler interessiert, kann mit dem Buch von ALFRED V. AHO – genannt das *Dragon-Book* – beginnen. Die Thematik geht über den Bau von Compilern hinaus und erstreckt sich ganz allgemein auf die Analyse und Verarbeitung von Zeichenfolgen.

1.1.3 Sprachenfamilien

Hat man eine Aufgabe, ein Problem zu lösen, so kann man drei Abschnitte auf dem Weg unterscheiden:

- Aufgabenstellung,
- Lösungsweg,
- Ergebnis.

Das Ergebnis ist nicht bekannt, sonst wäre die Aufgabe bereits gelöst. Die Aufgabenstellung und erforderlichenfalls einen Lösungsweg sollten wir kennen.

Mithilfe der bekannten Programmiersprachen von BASIC bis C++ beschreiben wir den Lösungsweg in einer für den Computer geeigneten Form. Diese Programmiersprachen werden als **algorithmische** oder **prozedurale** Programmiersprachen im weiteren Sinn bezeichnet, weil die Programme aus Prozeduren bestehen, die Anweisungen an den Computer enthalten (lateinisch *procedere* = vorangehen). Diese Familie wird unterteilt in die imperativen oder prozeduralen Sprachen im engeren Sinne einerseits und die objektorientierten Sprachen andererseits (lateinisch *imperare* = befehlen).

Bequemer wäre es jedoch, wir könnten uns mit der Beschreibung der Aufgabe begnügen und das Finden eines Lösungsweges dem Computer überlassen. Sein Nutzen würde damit bedeutend wachsen. Die noch nicht sehr verbreiteten **deklarativen** Programmiersprachen gehen diesen Weg (lateinisch *declarare* = erklären, beschreiben). Die Datenbank-Abfragesprache SQL (Structured Query Language) gehört hierher: in den Programmen (SQL-Script) steht, was man wissen will, nicht, wie man dazu kommt. Die deklarativen Sprachen unterteilt man in die **funktionalen** und die **logischen** oder **prädikativen** Sprachen.

Wir haben also folgende Einteilung (wobei die tatsächlich benutzten Sprachen Mischlinge sind und die Einordnung ihrem am stärksten ausgeprägten Charakterzug folgt):

- Prozedurale Sprachen im weiteren Sinn
  - imperative, algorithmische, operative oder im engeren Sinn prozedurale Sprachen (BASIC, FORTRAN, COBOL, C, PASCAL)
  - objektorientierte Sprachen (SMALLTALK, C++, Java)
- Deklarative Sprachen
  - funktionale oder applikative Sprachen (LISP, SCHEME, HASKELL)
  - logische oder prädikative Sprachen (PROLOG)

Diese Sprachentypen werden auch **Paradigmen** (Beispiel, Muster) genannt. Auf imperative und objektorientierte Sprachen gehen wir bald ausführlich ein. Zuerst ein kurzer Blick auf funktionale und prädikative Sprachen.

Programme in funktionalen Programmiersprachen wie LISP oder SCHEME bestehen aus Definitionen von Funktionen, äußerlich ähnlich einem Gleichungssystem, die auf Listen von Werten angewendet werden. Hier das Hello-World-Programm in LISP:

```
; LISP
(DEFUN HELLO-WORLD ()
                  (PRINT (LIST 'HELLO 'WORLD)))
```

Quelle 1.1 : LISP-Programm Hello, World

und auch noch in SCHEME:

```
(define hello-world
  (lambda ()
    (begin
      (write 'Hello-World)
      (newline)
      (hello-world))))
```

Quelle 1.2 : SCHEME-Programm Hello, World

Die großzügige Verwendung runder Klammern fällt ins Auge, aber ansonsten sind die Programme zu einfach, um die Eigenheiten der Sprachen zu erkennen. Die Sprache C ist trotz der Verwendung des Funktionsbegriffes keine funktionale Programmiersprache, da ihr Konzept nicht anders als in FORTRAN oder PASCAL auf der sequentiellen Ausführung von Anweisungen beruht.

Programmen in logischen oder prädikativen Sprachen wie PROLOG werden Fakten und Regeln zum Folgern mitgegeben, sie beantworten dann die Anfrage, ob eine Behauptung mit den Fakten und Regeln verträglich (wahr) ist oder nicht. Viele Denksportaufgaben legen eine solche Sprache nahe. Hier das Hello-World-Programm in PROLOG:

```
% HELLO WORLD.  Works with Sbp (prolog)

hello :-
printstring("HELLO WORLD!!!!").

printstring([]).
printstring([H|T]) :- put(H), printstring(T).
```

Quelle 1.3 : PROLOG-Programm Hello, World

Die Umgewöhnung von einem Paradigma auf ein anderes geht über das Erlernen einer neuen Sprache hinaus und beeinflusst die Denkweise, die Sicht auf ein Problem.

Es gibt ein zweite, von der ersten unabhängige Einteilung, die zugleich die historische Entwicklung spiegelt:

- maschinenorientierte Sprachen (Maschinensprache, Assembler)
- problemorientierte Sprachen (höhere Sprachen)

In der Frühzeit gab es nur die völlig auf die Hardware ausgerichtete und unbequeme Maschinensprache, wir haben eine Kostprobe gesehen. Assembler sind ein erster Schritt in Richtung auf die Probleme und die Programmierer zu. Höhere Sprachen wie FORTRAN sind von der Hardware schon ziemlich losgelöst und in diesem Fall an mathematische Probleme angepasst. Es gibt aber für spezielle Aufgaben wie Stringverarbeitung, Datenbankabfragen, Statistik oder Grafik Sprachen, die in ihrer Anpassung noch weiter gehen. Auch die zur Formatierung des vorliegenden Textes benutzte Sammlung von LaTeX-Makros stellt eine problemangepasste Sprache dar. Der Preis für die Erleichterungen ist ein Verlust an Allgemeinheit. Denken Sie an die Notensprache der Musik: an ihre Aufgabe gut angepasst, aber für andere Gebiete wie etwa die Chemie ungeeignet.

1.1.4 Imperative Programmiersprachen

Der Computer kennt nur Bits, das heißt Nullen und Einsen. Für den Menschen ist diese Ausdrucksweise unangebracht. Zum Glück sind die Zeiten, als man die Bits einzeln von Hand in die Lochstreifen schlug, vorbei.

Die nächste Stufe war die Zusammenfassung mehrerer Bits zu Gruppen, die man mit Buchstaben und Ziffern bezeichnen konnte. Ein Ausschnitt eines Programmes für die ZUSE Z 22 im Freiburger Code aus den fünfziger Jahren:

```
B15         Bringe den Inhalt von Register 15 in den Akku
U6          Kopiere den Akku nach Register 6
B18         Bringe den Inhalt von Register 18 in den Akku
+           Addiere Akku und Reg. 6, Summe in Akku und 6
B13         Bringe den Inhalt von Register 13 in den Akku
X           Multipliziere Akku mit Register 6
CGKU30+1    Kopiere den Akku nach der Adresse, die in
            Register 30 steht; inkrementiere Register 30
0           leere Operation
```

Quelle 1.4 : Ausschnitt aus einem Programm für die ZUSE Z 22

Man musste dem Computer in aller Ausführlichkeit sagen, was er zu tun hatte. Das war auch mühsam, aber diese Art der Programmierung gibt es heute noch unter dem Namen **Assemblerprogrammierung**. Man braucht sie, wenn man die Hardware fest im Griff haben will, also an den Grenzen Software - Hardware (Treiberprogramme). Darüber hinaus sind gute Assemblerprogramme schnell, weil sie nichts Unnötiges tun. Programmieren in Assembler setzt vertiefte Kenntnisse der Hardware voraus. Für PCs gibt es von Microsoft eine Kombination von Quick C mit Assembler, die es gestattet, das große Programm in der höheren Sprache C und einzelne kritische Teile in Assembler zu programmieren. Wer unbedingt den herben Reiz der Assemblerprogrammierung kennenlernen will, hat es mit dieser Kombination einfach.

Die meisten Programmierer wollen jedoch nicht Speicherinhalte verschieben, sondern Gleichungen lösen oder Wörter suchen[3]. Schon Mitte der fünfziger Jahre entstand daher bei der Firma IBM die erste höhere Programmiersprache, und zwar zum Bearbeiten mathematischer Aufgaben. Die Sprache war daher stark an die Ausdrucksweise der Mathematik angelehnt und zumindest für die mathematisch gebildete Welt einigermaßen bequem. Sie wurde als *formula translator*, abgekürzt **FORTRAN** bezeichnet. FORTRAN wurde im Laufe der Jahrzehnte weiter entwickelt – zur Zeit ist FORTRAN 90 bzw. 95 aktuell – und ist auch heute noch die in der Technik am weitesten verbreitete Programmiersprache. Kein Ingenieur kommt an FORTRAN vorbei. Ein Beispiel findet sich in Abschnitt 1.4.3 *Parameterübergabe* auf Seite 82.

[3]Recht betrachtet, will man auch keine Gleichungen, sondern Aufgaben wie die Dimensionierung eines Maschinenteils oder das Zusammenstellen eines Sachregisters lösen.

Die Kaufleute hatten mit Mathematik weniger am Hut, dafür aber große Datenmengen. Sie erfanden Ende der fünfziger Jahre ihre eigene Programmiersprache **COBOL**, das heißt *Common Business Oriented Language*. Dass Leutnant GRACE M. HOPPER (eine Frau, zuletzt im Admiralsrang) sowohl den ersten Bug erlegt wie auch COBOL erfunden habe, ist eine Legende um ein Körnchen Wahrheit herum. COBOL ist ebenfalls unverwüstlich und gilt heute noch als die am weitesten verbreitete Programmiersprache. Kein Wirtschaftswissenschaftler kommt an COBOL vorbei. Ein COBOL-Programm liest sich wie gebrochenes Englisch:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000300 DATE-WRITTEN.   02/05/96        21:04.
000400*       AUTHOR    BRIAN COLLINS
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500     DISPLAY "HELLO, WORLD." LINE 15 POSITION 10.
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

Quelle 1.5 : COBOL-Programm Hello, World

Als die Computer in die Reichweite gewöhnlicher Studenten kamen, entstand das Bedürfnis nach einer einfachen Programmiersprache für das Gröbste, kurzum nach einem *Beginners' All Purpose Symbolic Instruction Code*. JOHN KEMENY und THOMAS KURTZ vom Dartmouth College in den USA erfüllten 1964 mit **BASIC** diesen Bedarf. Der Gebrauch von BASIC gilt in ernsthaften Programmiererkreisen als anrüchig[4]. Richtig ist, dass es unzählige, miteinander unverträgliche BASIC-Dialekte gibt, dass BASIC die Unterschiede zwischen Betriebssystem und Programmiersprache verwischt und dass die meisten BASIC-Dialekte keine ordentliche Programmstruktur ermöglichen und daher nur für kurze Programme brauchbar sind. Richtig ist aber auch, dass moderne BASIC-Dialekte wie HP-BASIC oder QuickBASIC von Microsoft über alle Hilfsmittel zur Strukturierung verfügen und dass in keiner anderen gängigen Programmiersprache die Bearbeitung von Strings so einfach ist wie in BASIC[5]. In der Messwerterfassung ist es beliebt. Fazit: die Kenntnis von GW-BASIC auf dem PC reicht für einen Programmierer nicht aus, aber für viele Aufgaben ist ein modernes BASIC ein brauchbares Werkzeug.

Anfang der sechziger Jahre wurde **ALGOL 60** aufgrund theoretischer Überlegungen entwickelt und nach einer umfangreichen Überarbeitung als **ALGOL 68** veröffentlicht. Diese Programmiersprache ist nie in großem Umfang angewendet worden, spielte aber eine bedeutende Rolle als Wegbereiter für die heutigen Programmiersprachen beziehungsweise die heutigen Fassungen älterer Sprachen. Viele Konzepte gehen auf ALGOL zurück.

Ende der sechziger Jahre hatte sich das Programmieren vom Kunsthandwerk zur Wissenschaft entwickelt, und NIKLAUS WIRTH von der ETH Zürich brachte **PASCAL** heraus, um seinen Studenten einen anständigen Programmierstil anzugewöhnen. PASCAL ist eine strenge und logisch aufgebaute Sprache, daher gut zum Lernen geeignet. Turbo-PASCAL

[4]No programmers write in BASIC, after the age of 12.

[5]1964 bot keine andere Programmiersprache nennenswerte Möglichkeiten zur Verarbeitung von Strings.

von Borland ist auf PCs weit verbreitet. Ein PASCAL-Beispiel findet sich in Abschnitt 1.4.3 *Parameterübergabe* auf Seite 82. Eine Weiterentwicklung von PASCAL ist **MODULA**.

Die Sprache C wurde von BRIAN KERNIGHAN, DENNIS RITCHIE und KEN THOMPSON in den siebziger Jahren entwickelt, um das Betriebssystem UNIX damit portabel zu gestalten. Lange Zeit hindurch gab das Buch der beiden Erstgenannten den Standard vor[6]. In den Achtzigern hat das American National Standards Institute (ANSI) an einem Standard für C gearbeitet, dem alle neueren Compiler folgen (hinterherhinken). Das ANSI-Dokument wurde als Internationaler Standard ANSI/ISO/IEC 9899:1990 anerkannt. Ende der neunziger Jahre wurde eine erneute Überarbeitung als Internationaler Standard ANSI/ISO/IEC 9899:1999[7] veröffentlicht, der auch als C9X bezeichnet wird. Den Stand der Dinge erfährt man bei:

`http://anubis.dkuug.dk/JTC1/SC22/WG14/`

Erfahrungsgemäß dauert es einige Jahre, bis die Compiler einen neuen Standard voll unterstützen.

Das **ANSI-C** von 1990 ist im wesentlichen eine Übermenge von **K&R-C**; die Nachführung der Programme – wenn überhaupt erforderlich – macht keine Schwierigkeiten. ANSI-C kennt ein Schlüsselwort von K&R nicht mehr (`entry`) und dafür mehrere neue.

C ist allgemein verwendbar, konzentriert, lässt dem Programmierer große Freiheiten (*having the best parts of FORTRAN and assembly language in one place*) und führt in der Regel zu schnellen Programmen, da vielen C-Anweisungen unmittelbar Assembler-Anweisungen entsprechen (Maschinennähe). Die Sprache hat einen kleinen Kern (wenige Schlüsselwörter), Erweiterungen und Hardwareabhängigkeiten stecken in den Bibliotheken. C-Programme gelten als unübersichtlich, aber das ist eine Frage des Programmierstils, nicht der Sprache[8]. Auf UNIX-Systemen hat man mit C die wenigsten Schwierigkeiten. Für DOS-PCs gibt es von Microsoft das preiswerte Quick-C und aus dem GNU-Projekt einen kostenlosen C-Compiler im Quellcode und betriebsklar kompiliert.

Aus C hat BJARNE STROUSTRUP von 1979 bis 1989 eine Sprache **C++** entwickelt, die ebenfalls eine Übermenge von C bildet. Der Denkansatz (Paradigma) beim Programmieren in C++ weicht jedoch erheblich von C ab, so dass man eine längere Lernphase einplanen muss, mehr als bei einem Übergang von PASCAL nach C. Da sich ANSI-C und C++ gleichzeitig entwickelt haben, sind einige Neuerungen von C++ in ANSI-C eingeflossen, zum Beispiel das Prototyping. Ein ANSI-C-Programm sollte von jedem C++-Compiler verstanden werden; das Umgekehrte gilt nicht. Aktuell ist der Standard ANSI/ISO/IEC 14882:1998.

1.1.5 Objektorientierte Programmiersprachen

In dem Maß, wie die Hardware leistungsfähiger wurde, wagten sich die Programmierer an komplexere und umfangreichere Aufgaben heran. Dass große Aufgaben in kleinere Teilaufgaben untergliedert werden müssen, ist eine alltägliche Erfahrung und nicht auf Programme beschränkt. Die Strukturierung einer Aufgabe samt ihrer Lösung gewann an Bedeutung. Programmiersprachen wie C, die die Strukturierung eines Programms in Module (Funktionen, Prozeduren, Subroutinen) erleichtern, verbreiteten sich.

Um 1980 herum war die Komplexität wieder so angewachsen, dass nach neuen Wegen zu ihrer Bewältigung gesucht wurde. Außerdem hatte die Software als Kostenfaktor die Hardware überholt. Es galt, umfangreiche Programme schnell und preiswert herzustellen und dabei noch deren Zuverlässigkeit sicherzustellen, ähnlich wie heutzutage Autos produziert wer-

[6]Das vergleichsweise schlanke Buch von K&R ist die erste Lektüre, sobald man einfache C-Programme schreiben kann. Über die Beschreibung von C hinaus birgt es wertvolle allgemeine Hinweise zum Programmieren. Anmerkungen hat STEVE SUMMIT unter `http://www.eskimo.com/˜scs/cclass/knotes/top.html` veröffenlicht.

[7]Den Standard kann man sich bei `www.ansi.org` gegen 18 US-Dollar im pdf-Format herunterladen, rund 500 Seiten. Auf der Titelseite steht *Printed in the United States of America*, auf dem Rücktitel *Printed in Switzerland*. Die Wahrheit in meinem Fall: *Printed in Karlsruhe*. Hoffentlich geht das nicht so weiter.

[8] Es gibt einen International Obfuscated C Code Contest, einen Wettbewerb um das unübersichtlichste C-Programm, siehe Abschnitt 1.12 *Obfuscated C* auf Seite 171.

den. Zwei Schlagwörter kamen auf: **Objektorientierung** und **Software Engineering**. Entkleidet man sie der merkantilen Übertreibungen, bleibt immer noch ein brauchbarer Kern von Ideen übrig.

Der Typbegriff wurde zur **Klasse** erweitert. Eine Klasse enthält Variable und zugehörige Funktionen, die nun Methoden genannt werden. Klassen können im Gegensatz zum Typ vom Programmierer definiert werden. Sie bilden eine Hierarchie, wobei höhere Klassen Eigenschaften an niedrigere vererben. Klassen haben eine genau definierte Schnittstelle (Interface) zum Rest des Programms, ihr Innenleben bleibt verborgen. Was sie tun, ist bekannt, wie sie es tun, geht niemanden etwas an. Diese scharfe Trennung von Innen und Außen ist wesentlich für den Klassenbegriff. Was für C Funktionsbibliotheken sind, das sind für C++ Klassenbibliotheken. Die Programmierarbeit besteht zu einem großen Teil im Schreiben von Klassen. Wie eine Variable die Verwirklichung (Realisierung, Instantiierung) eines Typs ist, so ist ein **Objekt** eine Instanz einer Klasse. Von einer Klasse können beliebig viele Objekte abgeleitet werden. Klassen und deren Objekte sind die Bausteine eines objektorientierten Programms. C++ hieß anfangs C mit Klassen.

Neben C++ ist eine zweite objektorientierte Erweiterung von C entstanden, die unter dem Namen **Objective C** in Verbindung mit dem Betriebssystem NeXT eine gewisse Verbreitung gefunden hat. Der GNU-C-Compiler unterstützt sowohl C++ wie Objective C, ansonsten ist es ziemlich still geworden um diese Sprache.

Es kommen noch ein paar Dinge hinzu, um das Programmieren zu erleichtern, aber das Wesentliche am objektorientierten Programmieren ist, dass die Aufgabe nicht mehr in Module zerlegt wird, die aus Anweisungen bestehen, sondern in voneinander unabhängige Objekte, die sich Mitteilungen oder Botschaften schicken. Die Objektorientierung setzt bei der Aufgabenanalyse ein, nicht erst bei der Umsetzung in eine Programmiersprache (Codierung).

Wie verhält sich C++ zu C? Manche sagen, dass C++ eine neue, von C völlig unabhängige Sprache sei. Andere wiederum betrachten C++ als eine umfangreiche Erweiterung von C. Auf jeden Fall sind die Kenntnisse, die man beim Lernen von C erworben hat, auch in C++ nützlich und keinesfalls überholt. C++-Compiler übersetzen die meisten C-Programme anstandslos. Zu feineren Unterschieden und Unverträglichkeiten siehe DAVID. R. TRIBBLE:

http://david.tribble.com/text/cdiffs.htm

Für numerische Aufgaben ist C++ in der Universität Karlsruhe um eine Klassenbibliothek namens **C-XSC** (Extended Scientific Calculation) mit Datentypen wie komplexen Zahlen, Vektoren, Matrizen und Intervallen samt den zugehörigen Operationen ergänzt worden, siehe das Buch von RUDI KLATTE et al.

SMALLTALK ist eine von Grund auf neu entwickelte, streng objektorientierte Sprache[9], im Gegensatz zu C++. JAVA wurde von der Firma SUN entwickelt. Hier das Hello-World-Programm in JAVA (in C++ lernen wir es in Abschnitt 1.62 auf Seite 109 kennen):

```
class HelloWorld {
        public static void main (String args[]) {
        for (;;) {
                 System.out.print("Hello World ");
                 }
        }
}
```

Quelle 1.6 : JAVA-Programm Hello, World

Ähnlichkeiten zu C sind erkennbar, die JAVA-Entwickler waren vermutlich C-Programmierer.

Auf die übrigen 989 Programmiersprachen[10] soll aus Platzgründen nicht eingegangen werden. Braucht man überhaupt mehrere Sprachen? Einige Sprachen wie FORTRAN und

---

[9]SMALLTALK ist ungewohnt und sehr interessant, auch für LINUX verfügbar. Näheres siehe www.software.ibm.com/ad/smalltalk/, www.exept.de/, www.cetus-links.org/oo_smalltalk.html oder www.gsug.org/.

[10]Real programmers can write FORTRAN programs in any language.

COBOL sind historisch bedingt und werden wegen ihrer weiten Verbreitung noch lange leben. Andere Sprachen wie BASIC und C wenden sich an unterschiedliche Benutzerkreise. Wiederum andere eignen sich für spezielle Aufgaben besser als allgemeine Sprachen. Mit einer einzigen Sprache wird man auch in der Zukunft nicht auskommen. Die Schwierigkeiten beim Programmieren liegen im übrigen weniger in der Umsetzung in eine Programmiersprache – der Codierung – sondern in der Formulierung und Strukturierung der Aufgabe.

Was heißt, eine Sprache sei für ein System verfügbar? Es gibt einen Interpreter oder Compiler für diese Sprache auf diesem System (Hardware plus Betriebssystem). Die Bezeichnung *FORTRAN-Compiler für UNIX* reicht nicht, da es UNIX für verschiedene Hardware und zudem in verschiedenen Versionen gibt. Drei Dinge müssen zusammenpassen: Interpreter oder Compiler, Betriebssystem und Hardware.

1.1.6 Interpreter – Compiler – Linker

In höheren Programmiersprachen wie C oder FORTRAN geschriebene Programme werden als **Quellcode** (source code), Quellprogramm oder Quelltext bezeichnet. Mit diesem Quellcode kann der Computer unmittelbar nichts anfangen, er ist nicht ausführbar. Der Quellcode muss mithilfe des Computers und eines Übersetzungsprogrammes in **Maschinencode** übersetzt werden. Mit dem Maschinencode kann dann der Programmierer nichts mehr anfangen.

Es gibt zwei Arten von Übersetzern. **Interpreter** übersetzen das Programm jedesmal, wenn es aufgerufen wird. Die Übersetzung wird nicht auf Dauer gespeichert. Da der Quellcode zeilenweise bearbeitet wird, lassen sich Änderungen schnell und einfach ausprobieren. Andererseits kostet die Übersetzung Zeit. Interpreter findet man vorwiegend auf Home-Computern für BASIC, aber auch LISP-Programme, Shellscripts und awk-Scripts werden interpretiert.

Compiler übersetzen den Quellcode eines Programms als Ganzes und speichern die Übersetzung auf einem permanenten Medium. Zur Ausführung des Programms wird die Übersetzung aufgerufen. Bei der kleinsten Änderung muss das gesamte Programm erneut kompiliert werden, dafür entfällt die jedesmalige Übersetzung während der Ausführung. Compilierte Programme laufen also schneller ab als interpretierte. Es gibt auch Mischformen von Interpretern und Compilern, zum Beispiel für JAVA. Wie wir eingangs des Kapitels gesehen haben, arbeiten C- und C++-Compiler wie `cc(1)` und `CC(1)` in vier Durchgängen:

- Präprozessor
- eigentlicher Compiler (Übersetzung in Assembler-Code)
- Assembler (Übersetzung in Maschinen-Code)
- Linker

Der Präprozessor entfernt Kommentar und führt die Präprozessor-Anweisungen (siehe Abschnitt 1.9 *Präprozessor* auf Seite 129) aus. Ruft man den Compiler mit der Option `-P` auf, so erhält man die Ausgabe des Präprozessors in einer lesbaren Datei mit der Kennung `.i`.

Der eigentliche Compiler `ccom(1)` übersetzt den Quellcode in maschinenspezifischen, lesbaren Assemblercode. Die Compileroption `-S` liefert diesen Code in einer Datei mit der Kennung `.s`. Bei einem einfachen Programm sollte man sich einmal das Vergnügen gönnen und den Assemblercode anschauen.

Der Assembler ist ein zweiter Übersetzer, der Assemblercode in Maschinensprache übersetzt. Mit der Compileroption `-c` erhält man den Maschinencode (Objektcode, relocatable code) in einer nicht lesbaren Datei mit der Kennung `.o`.

Große Programme werden in mehrere Dateien aufgeteilt, die einzeln kompiliert werden, aber nicht einzeln ausführbar sind, weil erst das Programm als Ganzes einen Sinn ergibt. Das Verbinden der einzeln kompilierten Dateien zu einem ausführbaren Programm besorgt der Binder oder **Linker**. Die Compileroption `-c` unterdrückt das Linken und erzeugt eine nicht lesbare Datei mit der Kennung `.o`.

Unter UNIX werden üblicherweise Präprozessor, Compiler, Assembler und Linker von einem **Compilertreiber** aufgerufen, so dass der Benutzer nichts von den vier Schritten bemerkt. Man arbeitet mit dem Treiber `cc(1)`, `gcc(1)` oder `CC(1)` und erhält ein ausführbares Programm. Im Alltag meint man den Treiber, wenn man vom Compiler spricht.

Üblicherweise erzeugt ein Compiler Maschinencode für die Maschine, auf der er selbst läuft. **Cross-Compiler** hingegen erzeugen Maschinencode für andere Systeme. Das ist gelegentlich nützlich.

Der Name des Programms im C-Quellcode hat die Kennung `.c`, in FORTRAN und PASCAL entsprechend `.f` und `.p`. Das kompilierte, aber noch nicht gelinkte Programm wird als **Objektcode** oder **relozierbar** (relocatable) bezeichnet, der Dateiname hat die Kennung `.o` oder `.obj`. Das lauffähige Programm heißt **ausführbar** (executable), sein Name hat keine Kennung. Unter PC-DOS sind die Namen ausführbarer Programme durch `.com` oder `.exe` gekennzeichnet. Ein kompiliertes Programm wird auch **Binary** genannt, im Gegensatz zum Quelltext. Ein Programm ist **binär-kompatibel** zu einem anderen System, wenn es in seiner ausführbaren Form unter beiden läuft.

Hat sich ein Programm anstandslos kompilieren lassen und erzeugt beim Aufruf die Fehlermeldung `File not found`, dann liegt das fast immer daran, dass das Arbeitsverzeichnis nicht im Befehlspfad enthalten ist. Man ruft dann das neue Programm mit einem Punkt als Pfadangabe auf:

```
./myprogram
```

und veranlasst so die Shell, das Programm im Arbeitsverzeichnis zu suchen. Alternativ könnte man auch den Punkt in den Pfad aufnehmen.

Bei den Operanden spielt es eine Rolle, ob ihre Eigenschaften vom Übersetzer bestimmt werden oder von Programm und Übersetzer gemeinsam – zur Übersetzungszeit – oder während der Ausführung des Programmes – zur Laufzeit. Der zweite Weg wird als **statische Bindung** bezeichnet, der dritte als **dynamische Bindung**. Die Größe einer Ganzzahl (2 Bytes, 4 Bytes) ist durch den Compiler gegeben. Die Größe eines Arrays könnte im Programm festgelegt sein oder während der Ausführung berechnet werden. Es ist auch denkbar, aber in C nicht zugelassen, den Typ einer Variablen erst bei der Ausführung je nach Bedarf zu bestimmen.

Einen Weg zurück vom ausführbaren Programm zum Quellcode gibt es nicht. Das Äußerste ist, mit einem **Disassembler** aus dem ausführbaren Code Assemblercode zu erzeugen, ohne Kommentar und typografische Struktur. Nur bei kurzen, einfachen Programmen ist dieser Assemblercode verständlich.

1.1.7 Qualität und Stil

Unser Ziel ist ein gutes Programm. Was heißt das im einzelnen? Ein Programm soll selbstverständlich **fehlerfrei** sein in dem Sinn, dass es aus zulässigen Eingaben richtige Ergebnisse erzeugt. Außer in seltenen Fällen lässt sich die so definierte Fehlerfreiheit eines Programms nicht beweisen. Man kann nur – nach einer Vielzahl von Tests und längerem Gebrauch – davon reden, dass ein Programm zuverlässig ist, ein falsches Ergebnis also nur mit geringer Wahrscheinlichkeit auftritt.

Ein Programm soll **robust** sein, das heißt auf Fehler der Eingabe oder der Peripherie vernünftig reagieren, nicht mit einem Absturz. Das Schlimmste ist, wenn ein Programm trotz eines Fehlers ein scheinbar richtiges Ergebnis ausgibt. Die Fehlerbehandlung macht oft den größeren Teil eines Programmes aus und wird häufig vernachlässigt. Die Sprache C erleichtert diese Aufgabe.

Ein Programm ist niemals fertig und soll daher **leicht zu ändern** sein. Die Entdeckung von Fehlern, die Berücksichtigung neuer Wünsche, die Entwicklung der Hardware, Bestrebungen zur Standardisierung und Lernvorgänge der Programmierer führen dazu, dass Programme immer wieder überarbeitet werden. Kleinere Korrekturen werden durch **Patches** behoben, wörtlich Flicken. Das sind Ergänzungen zum Code, die nicht gleich eine neue Version rechtfertigen. Für manche Fehler lassen sich auch ohne Änderung des Codes **Umgehungen** finden, sogenannte Workarounds. Nach umfangreichen Änderungen – möglichst Verbesserungen – erscheint eine neue **Version** des Programmes. Ein Programm, von dem nicht einmal jährlich eine Überarbeitung erscheint, ist tot. Jede Woche eine neue Version ist natürlich auch keine Empfehlung. Leichte Änderbarkeit beruht auf Übersichtlichkeit, ausführlicher Dokumentation und Vermeidung von Hardwareabhängigkeiten. Die Übersichtlichkeit wiederum erreicht man durch eine zweckmäßige Strukturierung, verständliche Namenswahl

und Verzicht auf besondere Tricks einer Programmiersprache, die zwar erlaubt, aber nicht allgemein bekannt sind. Gerade C erlaubt viel, was nicht zur Übersichtlichkeit beiträgt.

Änderungen zu erleichtern kann auch heißen, Änderungen von vornherein zu vermeiden, indem man die Programmteile so allgemein wie mit dem Aufwand vereinbar gestaltet.

Effizienz ist immer gefragt. Früher bedeutete das vor allem sparsamer Umgang mit dem Arbeitsspeicher. Das ist heute immer noch eine Tugend, tritt aber hinter den vorgenannten Kriterien zurück. Die moderne Software scheint zur Unterstützung der Chiphersteller geschrieben zu werden. An zweiter Stelle kam Ausführungsgeschwindigkeit, trotz aller Geschwindigkeitssteigerungen der Hardware ebenfalls noch eine Tugend, wenn sie mit Einfachheit und Übersichtlichkeit einhergeht. Mit anderen Worten: erst ein übersichtliches Programm schreiben und dann nachdenken, ob man Speicher und Zeit einsparen kann.

Ein Programm soll **benutzerfreundlich** sein. Der Benutzer am Terminal will bei alltäglichen Aufgaben ohne das Studium pfundschwerer Handbücher auskommen und bei den häufigsten Fehlern Hilfe vom Bildschirm erhalten. Er will andererseits auch nicht mit überflüssigen Informationen und nutzlosen Spielereien belästigt werden. Der Schwerpunkt der Programmentwicklung liegt heute weniger bei den Algorithmen, sondern bei der Interaktion mit dem Benutzer. Für einen Programmierer ist es nicht immer einfach, sich in die Rolle eines EDV-Laien zu versetzen.

Schließlich ist daran zu denken, dass man ein Programm nicht nur für den Computer schreibt, sondern auch für andere Programmierer. Erstens kommt es oft vor, dass ein Programm von anderen weiterentwickelt oder ergänzt wird; zweitens ist ein Programm eine von mehreren Möglichkeiten, einen Algorithmus oder einen komplexen Zusammenhang darzustellen. Der Quellcode sollte daher leicht zu lesen, **programmiererfreundlich** sein. Fordern wir also menschenfreundliche Programme.

C lässt dem Programmierer viel Freiheit, mehr als PASCAL. Damit nun nicht jeder schreibt, wie ihm der Schnabel gewachsen ist, hat die Programmierergemeinschaft Regeln und Gebräuche entwickelt. Ein Verstoß dagegen beeindruckt den Compiler nicht, aber das Programm ist mühsam zu lesen. Der Beautifier `cb(1)` automatisiert die Einhaltung einiger dieser Regeln, weitergehende finden sich in:

- NELSON FORD, Programmer's Guide, siehe Anhang,
- B. W. KERNIGHAN, P. J. PLAUGER, Software Tools, siehe Anhang,
- ROB PIKE, Notes on Programming in C, `/pub/../pikestyle.ps` auf `ftp.ciw.uni-karlsruhe.de`
- Firmen-Richtlinien wie Nixdorf Computer C-Programmierrichtlinien (Hausstandard), 1985
- K. HENNING, Portables Programmieren in C – Programmierrichtlinien, verfasst 1993 vom Hochschuldidaktischen Zentrum und vom Fachgebiet Kybernetische Verfahren und Didaktik der Ingenieurwissenschaften der RWTH Aachen im Auftrag von sechs Chemiefirmen. Der Verbreitung dieser Richtlinien stehen leider ein Hinweis auf das Urheberrecht sowie ein ausdrückliches Kopierverbot entgegen.

Ein- und dieselbe Aufgabe kann – von einfachen Fällen abgesehen – auf verschiedene Weisen gelöst werden. Der eine bevorzugt viele kleine Programmblöcke, der andere wenige große. Einer arbeitet gern mit Menüs, ein anderer lieber mit Kommandozeilen. Einer schreibt einen langen Kommentar an den Programmanfang, ein anderer zieht kurze, in den Programmcode eingestreute Kommentare vor. Solange die genannten objektiven Ziele erreicht werden, ist gegen einen persönlichen **Stil** nichts einzuwenden. Le style c'est l'homme.

1.1.8 Programmiertechnik

Bei kurzen Programmen, wie sie in diesem Buch überwiegen, setzt man sich oft gleich an das Terminal und legt los. Besonders jugendliche BASIC-Programmierer neigen zu dieser Programmiertechnik. Wenn man sich das nicht schnellstens abgewöhnt, kommt man nicht weit. Um *wirkliche* Programme zu schreiben, muss man systematisch vorgehen und viel Konzeptpapier verbrauchen, ehe es ans Hacken geht. Es gibt mehrere Vorgehensweisen. Eine verbreitete sieht fünf Stufen vor (waterfall approach):

- Aufgabenstellung (Vorstudien, Analyse, Formulierung),
- Entwurf (Struktur, Anpassen an Werkzeuge wie `make(1)`, RCS, CVS),
- Umsetzung in eine Programmiersprache (Codierung, Implementation),
- Test (Fehlersuche, Prüfungen, Messungen, Vergleich mit Punkt 1),
- Betrieb und Pflege (Wartung, Updating, Fehler-Management).

Die Programmiersprache, die für den Anfänger im Vordergrund des Programmierens steht, kommt erst an dritter Stelle. Wenn die beiden vorangehenden Punkte schlecht erledigt worden sind, kann auch ein Meister in C/C++ nichts mehr retten.

Der Zeitbedarf der einzelnen Stufen ist schwierig abzuschätzen, da Kleinigkeiten manchmal fürchterlich aufhalten. Lassen wir Betrieb und Pflege als zeitlich unbegrenzt heraus, und nehmen wir an, dass das Schreiben der Dokumentation parallel erfolgt, so lassen sich ungefähr folgende Anteile als Ausgangswerte für eine Zeitplanung nehmen:

- Aufgabenanalyse 20 %,
- Entwurf 30 %,
- Codierung 20 %,
- Test 30 %.

Wer Softwareprojekte zu seinem Broterwerb macht, sollte ein Tagebuch oder Protokoll führen, um Erfahrungen auf dem Papier festzuhalten und sie beim nächsten Projekt zu verwerten.

Bei der Codierung rechnet man mit 60 Zeilen Programmcode (ohne Kommentar und Leerzeilen) pro Tag und Programmierer. Das sind zwei bis drei Seiten DIN A4 mit Kommentar und Leerzeilen. Gleichzeitig ist das die Obergrenze für ein Programmmodul (in C eine Funktion). Haben Sie für ihr Projekt 100 Arbeitstage Zeit und einen Programmierer, so ergeben sich 20 Arbeitstage ür die Codierung gleich 20 Modulen zu je 60 Zeilen Code. Das sind grobe Werte, aber sie reichen für eine erste Abschätzung aus.

Bei Texten kann man von einer Seite pro Tag ausgehen. Liegt das Rohmaterial samt allen Abbildungen fertig vor, kommt man auch auf zehn Seiten pro Tag. Umgekehrt können schwierige Rechnungen oder das Beschaffen exotischer Literatur ein Manuskript beliebig verzögern. Korrekturlesen, das Zusammenstellen eines Index und ähnliche ungeliebte Arbeiten kosten auch Zeit, unter Umständen Wochen.

Die Programmentwicklung vollzieht sich in der Praxis nicht so geradlinig, wie es der obige Plan vermuten lässt. Aus jeder Stufe kommen Rücksprünge in vorangegangene Stufen vor, man könnte auch von Rückkoppelungen sprechen. Dagegen ist nichts einzuwenden, es besteht jedoch eine Gefahr. Wenn man nicht Zwangsmaßnahmen ergreift – Schlussstriche zieht – erreicht das Programmierprojekt nie einen definierten Zustand. Programmierer verstehen das, Kaufleute und Kunden nicht. Gilt auch für Buchmanuskripte.

Der steigende Bedarf an Software und ihre wachsende Komplexität verlangen die Entwicklung von Programmierverfahren, mit denen durchschnittliche Programmierer zuverlässige Programme entwickeln. Auf geniale *Real Programmers* allein kann sich keine Firma verlassen. Die Entwicklung dieser Programmiertechnik (Software Engineering) ist noch nicht abgeschlossen.

1.1.9 Aufgabenanalyse und Entwurf

1.1.9.1 Aufgabenstellung

Die meisten Programmieraufgaben werden verbal gestellt, nicht in Form einer mathematischen Gleichung. Zudem sind sie anfangs oft pauschal abgefasst, da dem Aufgabensteller[11] Einzelheiten noch nicht klar sind.

[11] Real programmers know better than the users what they need.

Auf der anderen Seite benötigt der Computer eine eindeutige, ins einzelne gehende Anweisung, da er – anders als ein Mensch – fehlende Informationen nicht aufgrund seiner Erfahrung und des gesunden Menschenverstandes ergänzt.

Der erste Schritt bei der Programmentwicklung ist daher die **Formulierung** der Aufgabe. Zu diesem Schritt kehrt man im Verlauf des Programmierens immer wieder zurück, um zu ergänzen oder zu berichtigen. Es ist realistisch, für die Aufgabenanalyse rund ein Drittel des gesamten Zeitaufwandes anzusetzen. Die Aufgabe wird in einem **Pflichtenheft** schriftlich festgehalten, das zur Verständigung zwischen Entwickler und Anwender sowie der Entwickler untereinander dient. Fragen in diesem Zusammenhang sind:

- Welche Ergebnisse soll das Programm liefern?
- Welche Eingaben sind erforderlich?
- Welche Ausnahmefälle (Fehler) sind zu berücksichtigen?
- In welcher Form sollen die Ergebnisse ausgegeben werden?
- Wer soll mit dem Programm umgehen?
- Auf welchen Computern soll das Programm laufen?

Anfänger sehen die Schwierigkeiten des Programmierens in der Umsetzung des Lösungsweges in eine Programmiersprache, in der Codierung. Nach einigem Üben stellt sich dann heraus, dass die dauerhaften Schwierigkeiten in der Formulierung und Analyse der Aufgabe, allenfalls noch im Suchen nach Lösungen liegen, während die Codierung größtenteils Routine wird.

Nach unserer Erfahrung sollte man eine Aufgabe zunächst einmal so formulieren, wie sie den augenblicklichen Bedürfnissen entspricht. Dann sollte man sich mit viel Phantasie ausmalen, was alles noch dazu kommen könnte, wenn Geld, Zeit und Verstand keine Schranken setzen würden (*I have a dream ...*). Drittens streiche man von diesem Traum gnadenlos alles weg, was nicht unbedingt erforderlich und absolut minimal notwendig ist – ohne das vielleicht nur asymptotisch erreichbare Ziel aus den Augen zu verlieren. So kommt man mit beschränkten Mitteln zu Software, die sich entwickeln kann, wenn die Zeit dafür reif ist. Anpassungsfähigkeit ist für Software und Lebewesen wichtiger als Höchstleistungen.

1.1.9.2 Zerlegen in Teilaufgaben

Controlling complexity is the essence of computer programming (B. W. KERNIGHAN, P. J. PLAUGER, Software Tools). Komplexe Aufgaben werden in mehreren Stufen in **Teilaufgaben** zerlegt, die überschaubar sind und sich durch eine **Funktion** oder **Prozedur** im Programm lösen lassen. Insofern spiegelt die Zerlegung bereits die spätere **Programmstruktur**[12] wider. Das Hauptprogramm soll möglichst wenig selbst erledigen, sondern nur Aufrufe von Unterprogrammen enthalten und somit die große Struktur widerspiegeln. Oft ist folgende Gliederung ein zweckmäßiger Ausgangspunkt:

- Programmstart (Initialisierungen)
- Eingabe, Dialog
- Rechnung
- Ausgabe
- Hilfen
- Fehlerbehandlung
- Programmende, Aufräumen

Bei den Teilaufgaben ist zu fragen, ob sie sich – ohne die Komplexität wesentlich zu erhöhen – allgemeiner formulieren lassen. Damit lässt sich die Verwendbarkeit von Programmteilen verbessern. Diese Strategie wird als **Top-down-Entwurf** bezeichnet. Man geht vom Allgemeinen ins Einzelne.

[12]Real programmers disdain structured programming.

1.1.9.3 Zusammensetzen aus Teilaufgaben

Der umgekehrte Weg – **Bottom-up-Entwurf** – liegt nicht so nahe. Es gibt wiederkehrende **Grund-Operationen** wie Suchen, Sortieren, Fragen, Ausgeben, Interpolieren, Zeichnen eines Kreisbogens. Aus diesen lässt sich eine gegebene Aufgabe zu einem großen Teil zusammensetzen, so dass nur wenige spezielle Teilaufgaben übrig bleiben. Hat man die Grundoperationen einmal programmiert, so vereinfacht sich der Rest erheblich.

In praxi wendet man eine gemischte Strategie an. Man zerlegt die übergeordnete Aufgabe in Teilaufgaben, versucht diese in Grundoperationen auszudrücken und kommt dann wieder aufsteigend zu einer genaueren und allgemeiner gültigen Formulierung. Dieser Ab- und Aufstieg kann sich mehrmals wiederholen. Die Aufgabenstellung ist nicht unveränderlich. Genau so geht man bei der Planung von Industrieanlagen vor.

Man darf nicht den Fehler machen, die Aufgabe aus Bequemlichkeit den Eigenheiten eines Computers oder einer Programmiersprache anzupassen. Der Benutzer hat Anspruch auf ein gut und verständlich funktionierendes Programm. Die Zeiten, als der Computer als Entschuldigung für alle möglichen Unzulänglichkeiten herhalten musste, sind vorbei.

1.1.10 Prototyping

In dem häufig vorkommenden Fall, dass die Anforderungen an das Programm zu Beginn noch verschwommen sind, ist es zweckmäßig, möglichst rasch ein lauffähiges Grundgerüst, ein Skelett zu haben. Mit diesem kann man dann spielen und Erfahrungen sammeln in einem Stadium, in dem der Programmcode noch überschaubar und leicht zu ändern ist.

Bei einem solchen **Prototyp** sind nur die benutzernahen Funktionen halbwegs ausgebaut, die datennahen Funktionen schreiben vorläufig nur ihren Namen auf den Bildschirm. Von einem menugesteuerten Vokabeltrainer beispielsweise schreibt man zunächst das Menusystem und lässt die Funktionen, die die eigentliche Arbeit erledigen, leer oder beschränkt sie auf die Ausgabe ihres Namens. Damit liegt die **Programmstruktur** – das Knochengerüst – fest. Gleichzeitig macht man sich Gedanken über die **Datenstruktur**. Steht der Prototyp, nimmt man den **Datenaustausch** zwischen den Funktionen hinzu (Parameterübergabe und -rückgabe), immer noch mit Bildschirmmeldungen anstelle der eigentlichen Arbeit. Funktioniert auch das wie gewünscht, füllt man eine Funktion nach der anderen mit Code.

Diese Vorgehensweise lenkt die Entwicklung zu einem möglichst frühen Zeitpunkt in die gewünschte Richtung. Bei einem kommerziellen Auftrag bezieht sie den Auftraggeber in die Entwicklung ein und fördert das gegenseitige Verständnis, aber auch bei privaten Projekten verhindert sie, dass man viel Code für `/dev/null` schreibt.

Das Prototyping ist sicher nicht für alle Programmieraufgaben das beste Modell – es gibt auch noch andere Modelle – aber für dialogintensive kleine und mittlerer Anwendungen recht brauchbar und in C leicht zu verwirklichen.

1.1.11 Flussdiagramme

Programme werden schnell unübersichtlich. Man hat daher schon früh versucht, mit Hilfe grafischer Darstellungen[13] den Überblick zu behalten, aber auch diese neigen zum Wuchern. Ein grundsätzlicher Mangel ist die Beschränkung eines Blattes Papier auf zwei Dimensionen. Es ist unmöglich, ein umfangreiches Programm durch eine einzige halbwegs überschaubare Grafik zu beschreiben.

Flussdiagramme (flow chart), auch Blockdiagramme genannt, sollen die Abläufe innerhalb eines Programmes durch Sinnbilder nach DIN 66 001 und Text darstellen, unabhängig von einer Programmiersprache. Obwohl das Flussdiagramm vor dem Programmcode erstellt werden sollte, halten sich viele Programmierer nicht an diese Reihenfolge. Zum Teil ersetzt eine gute typografische Gestaltung der Programmquelle auch ein Flussdiagramm, während das Umgekehrte nicht gilt. Ein Flussdiagramm ist nicht mit einem Syntaxdiagramm zu verwechseln, lesen Sie die beiden entsprechenden Abbildungen, die die if-else-Verzweigung darstellen, einmal laut vor.

[13] Real programmers don't draw flowcharts.

true Bedingung **false**

Anweisung 1
Anweisung 2
Anweisung 3

Anweisung 4

Abb. 1.1: Flussdiagramm einer if-else-Verzweigung

if Bedingung

then **else**

Anweisung 1

Anweisung 2

Anweisung 3

Anweisung 4

Abb. 1.2: Nassi-Shneiderman-Diagramm einer if-else-Verzweigung

Nassi-Shneiderman-Diagramme oder Struktogramme nach ISAAC NASSI und BEN SHNEIDERMAN sind ein weiterer Versuch, den Programmablauf grafisch darzustellen. Sie sind näher an eine Programmiersprache angelehnt, so dass es leicht fällt, nach dem Diagramm eine Quelle zu schreiben. Das lässt sich teilweise sogar mit CASE-Werkzeugen in beide Richtungen automatisieren.

1.1.12 Memo Grundbegriffe

- Maschinen verstehen nur Maschinensprache, die hardwareabhängig und für Menschen unverständlich ist.
- Programmierer verwenden höhere, an die Aufgaben angepasste Programmiersprachen, die für Maschinen unverständlich sind. Was sie schreiben, wird Quelle (source) genannt.
- Übersetzer (Compiler, Interpreter) übersetzen Quelltexte höherer Programmiersprachen in Maschinensprache. Der umgekehrte Weg ist praktisch nicht gangbar.
- Deklarative Sprachen beschreiben die Aufgabe, prozedurale den Lösungsweg.
- Innerhalb der prozeduralen Sprachen gehören BASIC, FORTRAN, PASCAl, COBOL und C zum imperativen Zweig, JAVA, SMALLTALK und C++ zum objektorientierten.
- Die Objektorientierung ist ein Versuch, mit der wachsenden Komplexität der Programme fertig zu werden.
- Die Herstellung eines Programms beginnt mit einer gründlichen Analyse der Aufgabe. Die Umsetzung in eine Programmiersprache (Codierung) ist dann vergleichsweise harmlos.
- Ein Programm soll nicht nur die zugrundeliegende Aufgabe richtig lösen, sondern auch gegen Fehler und Ausnahmen unempfindlich (robust) sein. Die Fehlerbehandlung erfordert mehr Programmzeilen als die eigentliche Aufgabe.
- Ein Programm soll einfach zu ändern sein. Dies wird durch eine gute Struktur und reichlich Kommentar erleichtert (wenn man schon keine ausführliche Dokumentation schreibt).
- Ein Programm soll menschenfreundlich sein.

1.1.13 Übung Grundbegriffe

Nehmen wir an, der Weg zu Ihrem Arbeitsplatz bestehe aus mehreren Teilstrecken mit unterschiedlichen Gegebenheiten. Sie wollen wissen, was es bringt und kostet, wenn Sie einzelne Teilstrecken schneller oder langsamer zurücklegen.

Sie brauchen also ein Programm zur Analyse Ihres Arbeitsweges. Formulieren Sie die Aufgabe genauer, zerlegen Sie sie in Teilaufgaben, beschreiben Sie die Ein- und Ausgabe, berücksichtigen Sie Fehler des Benutzers. Aus welchen Größen besteht die Ausgabe, welche Eingaben sind für die Rechnungen erforderlich? Kann eine Division durch Null vorkommen? Das Ergebnis sollten einige Blätter Papier mit Worten, Formeln und Skizzen sein, nach denen ein Programmierer arbeiten könnte. Sie selbst sollen an dieser Stelle noch nicht an eine Programmiersprache denken.

Falls Ihnen die Übung zu einfach erscheint, machen Sie dasselbe für einen Vokabeltrainer, der außer Deutsch zwei Fremdsprachen beherrscht. Wortschatz anfangs je 1000 Vokabeln, erweiterbar. Erste Frage: Was gehört alles zu einer Vokabel?

1.2 Programmer's Workbench

Unter der *Werkbank des Programmierers* werden UNIX-Werkzeuge zusammengefaßt, die zum Programmieren benötigt werden. Auf Maschinen, die nicht zur Programmentwicklung eingesetzt werden, können sie fehlen. Das Werkzeug `make(1)` und die Revisionskontrolle sind auch bei Projekten außerhalb der Programmierung nützlich, vor allem beim Bearbeiten umfangreicher Manuskripte.

1.2.1 Nochmals die Editoren

Editoren wurden bereits im UNIX-Kapitel, Abschnitt **??** *Writer's Workbench* auf Seite **??** erläutert. Hier geht es nur um einige weitere Eigenschaften des Editors `vi(1)`, die beim Schreiben von Programmquellen von Belang sind.

Im Quellcode werden üblicherweise Schleifenrümpfe und dergleichen um eine Tabulatorbreite eingerückt, die als Default 8 Leerzeichen entspricht. Bei geschachtelten Schleifen stößt der Text schnell an den rechten Seitenrand. Es empfiehlt sich, in dem entsprechenden Verzeichnis eine Datei `.exrc` mit den Zeilen:

```
set tabstop=4
set showmatch
set number
```

anzulegen. Die Option `showmatch` veranlaßt den `vi(1)`, bei jeder Eingabe einer rechten Klammer kurz zur zugehörigen linken Klammer zu springen. Die Option `number` führt zur Anzeige der Zeilennummern, die jedoch nicht Bestandteil des Textes werden. Eine Zeile `set lisp` ist eine Hilfe beim Eingeben von LISP-Quellen.

Steht der Cursor auf einer Klammer, so läßt das Kommando `%` den Cursor zur Gegenklammer springen und dort verbleiben.

Auch beim `emacs(1)` gibt es einige Wege, das Schreiben von Quellen zu erleichtern, insbesondere natürlich, falls es um LISP geht. Der Editor `nedit(1)` lässt sich auf den Stil aller gängigen Programmiersprachen einschließlich LaTeX einstellen und ist in vielen Linux-Distributionen enthalten.

1.2.2 Compiler und Linker (cc, ccom, ld)

Auf das Schreiben der Quelltexte mit einem Editor folgt ihre Übersetzung in die Sprache der jeweiligen Maschine mittels eines Übersetzungsprogrammes, meist eines **Compilers**. Jedes vollständige UNIX-System enthält einen C-Compiler; Compiler für weitere Programmiersprachen sind optional. Auf unserer Anlage sind zusätzlich ein FORTRAN- und ein PASCAL-Compiler vorhanden, wobei von FORTRAN gegenwärtig die Versionen 77 und 90 nebeneinander laufen.

Kompilieren bedeutete vor der EDV-Zeit zusammentragen. Im alten Rom hatte es auch noch die Bedeutung von plündern. In unseren Herzensergießungen haben wir viel aus Büchern, Zeitschriften, WWW-Seiten und Netnews kompiliert.

Ein Compiler übersetzt den Quellcode eines Programmes in Maschinensprache. Die meisten Programme enthalten Aufrufe von externen Programmodulen, die bereits vorübersetzt und in Bibliotheken zusammengefaßt sind. Beispiele sind Ausgaberoutinen oder mathematische Funktionen. Der ausführbare Code dieser externen Module wird erst vom **Linker**[14] mit dem Programmcode vereinigt, so daß ein vollständiges ausführbares Programm entsteht. Es gibt die Möglichkeit, die externen Module erst zur Laufzeit hinzuzunehmen; das heißt **dynamisches Linken** und spart Speicherplatz. Dabei werden die Module entweder beim Laden des Programms in den Arbeitsspeicher oder erst bei ihrem Aufruf hinzugeladen (load on demand). Benutzen mehrere Programme ein in den Arbeitsspeicher kopiertes Modul gemeinsam anstatt jeweils eine eigene Kopie anzulegen, so kommt man zu den **Shared Libraries** und spart nochmals Speicherplatz.

Die Aufrufe lauten `cc(1)`, `f77(1)`, `f90(1)` und `pc(1)`. Diese Kommandos rufen **Compilertreiber** auf, die ihrerseits die eigentlichen Compiler `/lib/ccom`, `f77comp`, `f90comp` und `pascomp` starten und noch weitere Dinge erledigen. Ohne Optionen rufen die Compilertreiber auch noch den Linker `/bin/ld(1)` auf, so dass das Ergebnis ein lauffähiges Programm ist, das als Default den Namen `a.out(4)` trägt. Mit dem Namen `a.out(4)` sollte man nur vorübergehend arbeiten (mit `mv(1)` ändern). Der Aufruf des C-Compilers sieht beispielsweise so aus:

[14] Linker werden auch Binder, Mapper oder Loader genannt. Manchmal wird auch zwischen Binder und Loader unterschieden, soll uns hier nicht beschäftigen.

```
cc source.c
cc -g source.c -lm -L./lib -I. -DMAX=100
```

Die erste Zeile stellt den minimalen Aufruf dar, die zweite einen um gängige Optionen erweiterten. Die Option -g veranlaßt den Compiler, zusätzliche Informationen für den symbolischen Debugger zu erzeugen. Weitere Optionen sind:

- -v (verbose) führt zu etwas mehr Bemerkungen beim Übersetzen,
- -o (output) benennt die ausführbare Datei mit dem auf die Option folgenden Namen, meist derselbe wie die Quelle, nur ohne Kennung: cc -o myprogram myprogram.c,
- -c hört vor dem Linken auf, erzeugt Objektfile mit der Kennung .o,
- -p (profile) erzeugt beim Ablauf des Programmes eine Datei mon.out, das mit dem Profiler prof(1) ausgewertet werden kann, um Zeitinformationen zum Programm zu erhalten,
- -O optimiert das ausführbare Programm oder auch nicht.

Der Quelltext des C-Programmes steht in der Datei source.c, die einen beliebigen Namen tragen kann, nur sollte der Name mit der Kennung .c enden. Die anschließende Option -lm fordert den Linker auf, die mathematische Standard-Bibliothek einzubinden. Die Option -L./lib wendet sich ebenfalls an den Linker und teilt ihm mit, dass sich im Verzeichnis ./lib weitere Bibliotheken befinden. Die Reihenfolge, in der Bibliotheken eingebunden werden, ist wichtig. Die Option -I. veranlasst den Präprozessor, Include-Dateien auch im aktuellen Verzeichnis zu suchen, was er nicht immer automatisch tut. Es könnte auch ein anderes Verzeichnis angegeben werden. Die Option -DMAX=100 definiert eine symbolische Konstante namens MAX und weist ihr den Wert 100 zu, genau wie eine Zeile:

```
#define MAX 100
```

im Quelltext, nur eben hier mit der Möglichkeit, den Wert bei der Übersetzung zu bestimmen. Speichermodelle wie unter PC-DOS gibt es in UNIX nicht. Hat man Speicher, kann man ihn uneingeschränkt nutzen.

Für C-Programme gibt es einen **Syntax-Prüfer** namens lint(1), den man unbedingt verwenden sollte. Er reklamiert nicht nur Fehler, sondern auch Stilmängel. Manchmal beanstandet er auch Dinge, die man bewußt gegen die Regeln geschrieben hat. Man muß seinen Kommentar sinnvoll interpretieren. Aufruf:

```
lint mysource.c
```

Ein verbesserter lint, ein *Secure Programming Lint* findet sich bei der University of Virginia unter:

```
http://www.splint.org/
```

Unter Linux ist lint(1) nicht überall vorhanden, dann kann man den Compiler gcc(1) mit einer Option aufrufen, die ihn nur zu einer Prüfung der Syntax veranlasst:

```
gcc -fsyntax-only -pedantic -Wall mysource.c
```

Ferner gibt es unter einigen UNIXen für C-Quelltexte einen **Beautifier** namens cb(1), der den Text in eine standardisierte Form mit Einrückungen usw. bringt und die Lesbarkeit erleichtert:

```
cb source.c > source.b
```

Wenn man mit dem Ergebnis source.b zufrieden ist, löscht man die ursprüngliche Datei source.c und benennt source.b in source.c um.

1.2.3 Unentbehrlich (make)

Größere Programme sind stark gegliedert und auf mehrere bis viele Dateien und Verzeichnisse verteilt. Der Compileraufruf wird dadurch länglich, und die Wahrscheinlichkeit, etwas zu vergessen, steigt. Hier hilft `make(1)`. Man schreibt einmal alle Angaben für den Compiler in ein `makefile` (auch `Makefile`) und ruft dann zum Kompilieren nur noch `make(1)` auf. Für Manuskripte ist `make(1)` ebenfalls zu gebrauchen. Statt Makefiles ließen sich auch Shellskripte einsetzen, die Stärke von `make(1)` liegt jedoch im Umgang mit Dateien unter Beachtung des Zeitstempels `mtime` (jüngster schreibender Zugriff). Werkzeuge wie `make(1)` werden als *Builder* bezeichnet.

Man lege für das Projekt ein eigenes Unterverzeichnis an, denn `make(1)` sucht zunächst im Arbeits-Verzeichnis. Das `makefile` beschreibt die Abhängigkeiten (dependencies) der Programmteile voneinander und enthält die Kommandozeilen zu ihrer Erzeugung. Ein einfaches `makefile` sieht so aus (Zeilen mit Kommandos müssen durch einen Tabulatorstop – *nicht* durch Spaces – eingerückt sein):

```
pgm:  a.o  b.o
        cc  a.o  b.o  -o pgm
a.o:  incl.h  a.c
        cc  -c a.c
b.o:  incl.h  b.c
        cc  -c  b.c
```

Quelle 1.7 : Einfaches Makefile

und ist folgendermaßen zu verstehen:

- Das ausführbare Programm (Ziel, Target) namens `pgm` hängt ab von den Modulen im Objektcode `a.o` und `b.o`. Es entsteht durch den Compileraufruf `cc a.o b.o -o pgm`.
- Das Programmodul `a.o` hängt ab von der include-Datei `incl.h` und dem Modul im Quellcode `a.c`. Es entsteht durch den Aufruf des Compilers mit `cc -c a.c`. Die Option `- c` unterbindet das Linken.
- Das Programmodul `b.o` hängt ab von derselben include-Datei und dem Modul im Quellcode `b.c`. Es entsteht durch den Compileraufruf `cc -c b.c`.

Ein `makefile` ist ähnlich aufgebaut wie ein Backrezept: erst werden die Zutaten aufgelistet, dann folgen die Anweisungen. Zu beachten ist, daß man am Ziel startet und rückwärts bis zu den Quellen geht. Kommentar beginnt mit einem Doppelkreuz und reicht bis zum Zeilenende. Leerzeilen werden ignoriert.

`make(1)` verwaltet auch verschiedene Versionen der Programmodule und paßt auf, daß eine neue Version in alle betroffenen Programmteile eingebunden wird. Umgekehrt wird eine aktuelle Version eines Moduls nicht unnötigerweise kompiliert. Warum wird im obigen Beispiel die include-Datei `incl.h` ausdrücklich genannt? Der Compiler weiß doch auf Grund einer entsprechenden Zeile im Quelltext, daß diese Datei einzubinden ist? Richtig, aber `make(1)` muß das auch wissen, denn die include-Datei könnte sich ändern, und dann müssen alle von ihm abhängigen Programmteile neu übersetzt werden. `make(1)` schaut nicht in die Quellen hinein, sondern nur auf die Zeitstempel (`mtime`) der Zutaten. Unveränderliche include-Dateien wie `stdio.h` brauchen nicht im `makefile` aufgeführt zu werden.

Nun ein etwas umfangreicheres Beispiel, das aber längst noch nicht alle Fähigkeiten von `make(1)` ausreizt:

```
# Kommentar, wie ueblich

CC = /bin/cc
CFLAGS =
FC = /usr/bin/f77
LDFLAGS = -lcl

all: csumme fsumme clean
```

```
csumme: csumme.c csv.o csr.o
        $(CC) -o csumme csumme.c csv.o csr.o

csv.o: csv.c
        $(CC) -c csv.c

csr.o: csr.c
        $(CC) -c csr.c

fsumme: fsumme.c fsr.o
        $(CC) -o fsumme fsumme.c fsr.o $(LDFLAGS)

fsr.o: fsr.f
        $(FC) -c fsr.f

clean:
        rm *.o
```

Quelle 1.8 : Makefile mit Makros und Dummy-Zielen

Zunächst werden einige Makros definiert, z. B. der Compileraufruf `CC`. Überall, wo im Makefile das Makro mittels `$(CC)` aufgerufen wird, wird es vor der Ausführung wörtlich ersetzt. Auf diese Weise kann man einfach einen anderen Compiler wählen, ohne im ganzen Makefile per Editor ersetzen zu müssen. Dann haben wir ein Dummy-Ziel `all`, das aus einer Aufzählung weiterer Ziele besteht. Mittels `make all` wird dieses Dummy-Ziel erzeugt, d. h. die aufgezählten Ziele. Unter diesen befindet sich auch eines namens `clean`, das ohne Zutaten daherkommt und offenbar nur bestimmte Tätigkeiten wie das Löschen temporärer Dateien bezweckt. Ein Dummy-Ziel ist immer out-of-date, die zugehörigen Kommandos werden immer ausgeführt.

`make(1)` darf rekursiv aufgerufen werden, ein Makefile darf `make`-Aufrufe enthalten, die sich auf *andere* Makefiles beziehen. Das kann so aussehen:

```
......
        cd subdirectory ; make all ; make clean
......
```

Gelangt `make` an diese Zeile, springt es in das Unterverzeichnis, sucht dort ein Makefile und erzeugt die Ziele `all` und `clean`. Anschließend macht es im Makefile des aktuellen Verzeichnisses weiter. Eine Anwendung ist ein aus mehreren Kapiteln bestehendes Skriptum, das komplett als Report formatiert werden soll, dazu noch die Kapitel in jeweils einem eigenen Unterverzeichnis als Artikel samt Folien. Natürlich sind die Makefiles in den Unterverzeichnissen weiche Links auf ein einziges Makefile. Ohne `make(1)` und eine entsprechende Verzeichnisstruktur würde man sich dumm und dämlich tippen.

Im GNU-Projekt wird Software im Quellcode für verschiedene Systeme veröffentlicht. In der Regel muß man die Quellen auf der eigenen Anlage kompilieren. Infolgedessen gehören zu den GNU-Programmen fast immer umfangreiche Makefiles oder sogar Hierarchien davon. Übung im Gebrauch von `make(1)` erleichtert die Einrichtung von GNU-Software daher erheblich. Oft wird ein an das eigene System angepaßtes Makefile erst durch ein Kommando `./configure` erzeugt. Die Reihenfolge bei solchen Programmeinrichtungen lautet dann:

```
./configure
(vi Makefile)
make
make install
make clean
```

wobei `make install` Schreibrechte in den betroffenen Verzeichnissen erfordert, also meist Superuserrechte. Gelegentlich wird `make(1)` aus einem Shellskript heraus aufgerufen, das einige Dinge vorbereitet. So wird zum Beispiel `sendmail(1)` durch den Aufruf des mitgelieferten Shellskripts `Build` erzeugt.

Das Skript `configure` erlaubt oft die Option `–prefix=DIR`, wobei `DIR` das Verzeichnis ist, in dem das ganze Gerödel eingerichtet werden soll, defaultmäßig meist `/usr/local`, aber manchmal besser `/usr` oder `/opt`. Da von `configure` alles Weitere abhängt, sollte man sich die zugehörige Protokoll-Datei `config.log` ansehen, auch wenn anscheinend keine Probleme aufgetreten sind.

Statt `make clean` kann man auch `make distclean` versuchen, das räumt noch gründlicher auf, so daß hinterher wieder mit `./configure` ein Neubeginn möglich ist.

1.2.4 Debugger (xdb, gdb)

Programme sind Menschenwerk und daher fehlerhaft[15]. Es gibt keine Möglichkeit, die Fehlerfreiheit eines Programmes festzustellen oder zu beweisen außer in trivialen oder idealen Fällen.

Die Fehler lassen sich in drei Klassen einteilen. Verstöße gegen die Regeln der jeweiligen Programmiersprache heißen **Grammatikfehler** oder **Syntaxfehler**. Sie führen bereits zu einem Abbruch des Kompiliervorgangs und lassen sich schnell lokalisieren und beheben. Der C-Syntax-Prüfer `lint` ist das beste Werkzeug zu ihrer Entdeckung. `wihle` statt `while` wäre ein einfacher Syntaxfehler. Fehlende oder unpaarige Klammern sind auch beliebt, deshalb enthält der `vi(1)` eine Funktion zur Klammerprüfung. Unzulässige Operationen mit Pointern sind ebenfalls an der Tagesordnung. Geht es um Texte, so fallen Tippfehler und Grammatikfehler in diese Klasse.

Falls das Programm die Kompilation ohne Fehlermeldung hinter sich gebracht hat, startet man es. Dann melden sich die **Laufzeitfehler**, die unter Umständen nur bei bestimmten und womöglich seltenen Parameterkonstellationen auftreten. Ein typischer Laufzeitfehler ist die Division durch eine Variable, die manchmal den Wert Null annimmt. Die Fehlermeldung lautet *Floating point exception*. Ein anderer häufig vorkommender Laufzeitfehler ist die Überschreitung von Arraygrenzen oder die Verwechslung von Variablen und Pointern, was zu einem *Memory fault*, einem Speicherfehler führt.

Die dritte Klasse bilden die **logischen Fehler** oder **Denkfehler**. Sie werden auch **semantische Fehler** genannt. Das Programm arbeitet einwandfrei, nur tut es nicht das, was sich der Programmierer vorgestellt hat. Ein typischer Denkfehler ist das Verzählen bei den Elementen eines Arrays oder bei Schleifendurchgängen um genau eins. Hier hilft der Computer nur wenig, da der Ärmste ja gar nicht weiß, was sich der Programmierer vorstellt. Diese Fehler kosten viel Mühe, doch solcherlei Verdrüsse pflegen die Denkungskräfte anzuregen, meint WILHELM BUSCH und hat recht.

Eine vierte Fehlerklasse liegt fast schon außerhalb der Verantwortung des Programmierers. Wenn das mathematische **Modell** zur Beschreibung eines realen Problems ungeeignet ist, mag das Programm so fehlerarm sein wie es will, seine Ergebnisse gehen an der Wirklichkeit vorbei. Für bestimmte Zwecke ist eine Speisekarte ein brauchbares Modell einer Mahlzeit, für andere nicht.

In diese Klasse fallen auch Fehler, die dadurch entstehen, dass wir im Computer stets mit Zahlen endlicher Länge rechnen, während in Wirklichkeit die Zahl π unendlich viele Dezimalstellen hat und gemessene Größen statistischen Schwankungen unterliegen, also unscharf begrenzte Intervalle darstellen. Grundkenntnisse in moderner numerischer Mathematik bewahren vor blindem Glauben an den Computer.

Ob und wie inhaltliche Fehler in Texten – falsche oder fehlende Angaben – einer der vorstehenden Klassen sinnvoll zugeordnet werden können, ist noch zu überlegen. Dann gibt es noch den Fehler *Thema verfehlt*, der vielleicht zum Modell-Fehler passt. Hintergrund dieser Gedanken ist die Anwendung von Software-Werkzeugen in Text-Projekten, was erwiesenermaßen die Arbeit erleichtert.

[15] Es irrt der Mensch, so lang er strebt. GOETHE, Faust. Oder *errare humanum est*, wie wir Lateiner sagen. Noch etwas älter: *αμαρτωλαι εν ανθρωποισιν επονται θνητοις*. Die entsprechende Aussage in babylonischer Keilschrift aus dem Codex Kombysis können wir leider aus Mangel an einem TeX-Font vorläufig nicht wiedergeben. In der nächsten Auflage werden wir jedoch eine eingescannte Zeichnung aus der Höhle von Rienne-Vaplus zeigen, die als älteste Dokumentation obiger Weisheit gilt.

Ein Fehler wird im Englischen auch als *bug* bezeichnet, was soviel wie Wanze oder Laus bedeutet. Ein Programm zu entlausen heißt Debugging. Dazu braucht man einen Debugger (déverminateur, déboguer). Das sind Programme, unter deren Kontrolle das verlauste Programm abläuft. Man hat dabei vielfältige Möglichkeiten, in den Ablauf einzugreifen. Ein **absoluter Debugger** wie der `adb(1)` bezieht sich dabei auf das lauffähige Programm im Arbeitsspeicher – nicht auf den Quellcode – und ist somit für die meisten Aufgaben wenig geeignet. Ein **symbolischer Debugger** wie der `sdb(1)`, der GNU `gdb(1)` oder der `xdb(1)` bezieht sich auf die jeweilige Stelle im Quelltext[16]. Debugger sind mächtige und hilfreiche Werkzeuge. Manche Programmierer gehen so weit, daß sie das Schreiben eines Programms als Debuggen einer leeren Datei bzw. eines weißen Blattes Papier ansehen. In der Übung wird eine einfache Anwendung des Debuggers vorgeführt.

Falls Sie auch mit dem UNIX-Debugger nicht alle Würmer in Ihrem Programm finden und vertreiben können, möchten wir Ihnen noch ein altes Hausrezept verraten, das aus einer Handschrift des 9. Jahrhunderts stammt. Das Rezept ist im Raum Wien – München entstanden und unter den Namen *Contra vermes* oder *Pro nescia* bekannt. Leider ist die `README`-Datei, welche die Handhabung erklärt, verlorengegangen. Wir schlagen vor, die Zeilen als Kommentar in das Programm einzufügen. Hier der Text:

> Gang út, nesso, mid nigun nessiklinon,
> ût fana themo marge an that bên,
> fan thêmo bêne an that flêsg,
> ût fan themo flêsgke an thia hûd,
> ût fan thera hûd an thesa strâla.
> Drohtin. Uuerthe sô!

1.2.5 Profiler (time, gprof)

Profiler sind ebenfalls Programme, unter deren Kontrolle ein zu untersuchendes Programm abläuft. Ziel ist die Ermittlung des Zeitverhaltens in der Absicht, das Programm schneller zu machen. Ein einfaches UNIX-Werkzeug ist `time(1)`:

```
time prim 1000000
```

Die Ausgabe sieht so aus:

```
real    0m 30.65s
user    0m 22.53s
sys     0m  1.07s
```

und bedeutet, daß die gesamte Laufzeit des Programms `prim` 30.65 s betrug, davon entfielen 22.53 s auf die Ausführung von Benutzeranweisungen und 1.07 s auf Systemtätigkeiten. Die Ausgabe wurde durch einen Aufruf des Primzahlenprogramms aus dem Skriptum *Programmieren in C/C++* erzeugt, das selbst Zeiten mittels des Systemaufrufs `time(2)` misst und rund 22 s für die Rechnung und 4 s für die Bildschirmausgabe meldet.

Ein weiterer Profiler ist `gprof(1)`. Seine Verwendung setzt voraus, daß das Programm mit der Option `-G` kompiliert worden ist. Es wird gestartet und erzeugt neben seiner normalen Ausgabe eine Datei `gmon.out`, das mit `gprof(1)` betrachtet wird. Besser noch lenkt man die Ausgabe von `gprof(1)` in eine Datei um, die sich lesen und editieren läßt:

```
gprof prim > prim.gprofile
```

Eine stark gekürzte Analyse mittels `gprof(1)` sieht so aus:

```
%time   the percentage of the total running time of the
        program used by this function.

cumsecs a running sum of the number of seconds accounted
        for by this function and those listed above it.
```

[16] Real programmers don't use source language debuggers.

```
seconds the number of seconds accounted for by this
        function alone.  This is the major sort for this
        listing.

calls   the number of times this function was invoked, if
        this function is profiled, else blank.

name    the name of the function.  This is the minor sort
        for this listing.

%time cumsecs seconds  calls msec/call name
 52.1   12.18   12.18                  $$remU
 22.2   17.38    5.20                  $$mulU
 20.8   22.25    4.87 333332      0.01 ttest
  2.1   22.74    0.49   9890      0.05 _doprnt
  0.8   22.93    0.19                  _mcount
  0.6   23.08    0.15                  $$divide_by_const
  0.6   23.22    0.14      1    140.00 main
  0.3   23.29    0.07   9890      0.01 _memchr
  0.2   23.34    0.05                  _write_sys
  0.1   23.36    0.02   9890      0.00 _printf
  0.0   23.37    0.01   9887      0.00 _write
  0.0   23.38    0.01   9887      0.00 _xflsbuf
  0.0   23.39    0.00   9890      0.00 _wrtchk
  0.0   23.39    0.00      1      0.00 _sscanf
  0.0   23.39    0.00      1      0.00 _start
  0.0   23.39    0.00      1      0.00 _strlen
  0.0   23.39    0.00      1      0.00 atexit
  0.0   23.39    0.00      1      0.00 exit
  0.0   23.39    0.00      1      0.00 ioctl
```

Wir sehen, daß die Funktion `ttest()` sehr oft aufgerufen wird und 4,87 s verbrät. Die beiden ersten Funktionen werden vom Compiler zur Verfügung gestellt (Millicode aus `/usr/lib/milli.a`) und liegen außerhalb unserer Reichweite.

Für genauere Auskünfte zieht man den Systemaufruf `times(2)`, den Debugger oder das UNIX-Kommando `prof(1)` in Verbindung mit der Subroutine `monitor(3)` heran.

1.2.6 Archive, Bibliotheken (ar)

Viele Teilaufgaben in den Programmen wiederholen sich immer wieder. Das sind Aufgaben, die mit dem System zu tun haben, Befehle zur Bildschirmsteuerung, mathematische Berechnungen wie Logarithmus oder trigonometrische Funktionen, Datenbankfunktionen oder Funktionen zur Abfrage von Meßgeräten am Bus.

Damit man diese Funktionen nicht jedesmal neu zu erfinden braucht, werden sie in **Bibliotheken** gepackt, die dem Programmierer zur Verfügung stehen. Teils stammen sie vom Hersteller des Betriebssystems (also ursprünglich AT&T), teils vom Hersteller der Compiler (bei uns Hewlett-Packard und GNU) oder der Anwendungssoftware, teils von Benutzern. Bibliotheken enthalten Programmbausteine, es lassen sich aber auch andere Dateien (Texte, Grafiken) in gleicher Weise zusammenfassen. Dann spricht man allgemeiner von **Archiven**. Außer den Dateien enthalten Archive Verwaltungsinformationen (Index) zum schnellen Finden der Inhalte. Diese Informationen wurden früher mit dem Kommando `ranlib(1)` eigens erzeugt, heute erledigt `ar(1)` das mit. Die Verwendung von Bibliotheken beim Programmieren wird im Skriptum *Programmieren in C/C++* erläutert.

Außer den mit dem Compiler gelieferten Bibliotheken kann man zusätzlich erworbene oder selbst erstellte Bibliotheken verwenden. Im Handel sind beispielsweise Bibliotheken mit Funktionen für Bildschirmmasken, zur Verwaltung index-sequentieller Dateien, für Grafik,

zur Meßwerterfassung und -aufbereitung und für besondere mathematische Aufgaben. Auch aus dem Netz laufen Bibliotheken zu. Eigene Bibliotheken erzeugt man mit dem UNIX-Kommando `ar(1)`; das Datei-Format ist unter `ar(4)` beschrieben. Ein Beispiel zeige den Gebrauch. Wir haben ein Programm `statistik.c` zur Berechnung von Mittelwert und Varianz der in der Kommandozeile mitgegebenen ganzen Zahlen geschrieben:

```
/* Statistische Auswertung von eingegebenen Werten
   Privat-Bibliothek ./libstat.a erforderlich
   Compileraufruf cc statistik.c -L . -lstat
*/

#define  MAX 100        /* max. Anzahl der Werte */
#include <stdio.h>

void exit(); double mwert(), varianz();

main(int argc, char *argv[])

{
int i, a[MAX];

if (argc < 3) {
    puts("Zuwenig Werte"); exit(-1);
}

if (argc > MAX + 1) {
    puts("Zuviel Werte"); exit(-1);
}

/* Uebernahme der Werte in ein Array */

a[0] = argc - 1;

for (i = 1; i < argc; i++) {
    sscanf(argv[i], "%d", a + i);
}

/* Ausgabe des Arrays */

for (i = 1; i < argc; i++) {
    printf("%d\n", a[i]);
}

/* Rechnungen */

printf("Mittelwert: %f\n", mwert(a));
printf("Varianz:    %f\n", varianz(a));

return 0;
}
```

Quelle 1.9 : C-Programm Statistik mit Benutzung einer eigenen Funktionsbibliothek

Das Programm verwendet die Funktionen `mwert()` und `varianz()`, die wir aus einer hausgemachten Funktionsbibliothek namens `libstat.a` entnehmen. Der im Kommentar genannte Compileraufruf mit der Option `-L .` veranlaßt den Linker, diese Bibliothek im Arbeits-Verzeichnis zu suchen. Die Funktionen sehen so aus:

```
double mwert(x)
int *x;
{
int j, k;
double m;
```

```
for (j = 1, k = 0; j <= *x; j++) {
    k = k + x[j];
}
m = (double)k / (double)*x;
return m;
}
```

Quelle 1.10 : C-Funktion Mittelwert ganzer Zahlen

```
extern double mwert();

double varianz(x)
int *x;
{
int j;
double m, s, v;

m = mwert(x);

for (j = 1, s = 0; j <= *x; j++) {
s = s + (x[j] - m) * (x[j] - m);
}
v = s / (*x - 1);
return v;
}
```

Quelle 1.11 : C-Funktion Varianz ganzer Zahlen

Diese Funktionen werden mit der Option `-c` kompiliert, so daß wir zwei Objektfiles `mwert.o` und `varianz.o` erhalten. Mittels des Aufrufes

```
ar -r libstat.a mwert.o varianz.o
```

erzeugen wir die Funktionsbibliothek `libstat.a`, auf die mit der Compileroption `-lstat` zugegriffen wird. Der Vorteil der Bibliothek liegt darin, daß man sich nicht mit vielen einzelnen Funktionsfiles herumzuschlagen braucht, sondern mit der Compileroption gleich ein ganzes Bündel verwandter Funktionen erwischt. In das Programm eingebunden werden nur die Funktionen, die wirklich benötigt werden.

Merke: Ein Archiv ist weder verdichtet noch verschlüsselt. Dafür sind andere Werkzeuge (`gzip(1)`, `crypt(1)`) zuständig.

1.2.7 Weitere Werkzeuge

Das Werkzeug `cflow(1)` ermittelt die Funktionsstruktur zu einer Gruppe von C-Quell- und Objektfiles. Der Aufruf:

```
cflow statistik.c
```

liefert auf `stdout`

```
1 main: int(), <statistik.c 15>
2 puts: <>
3 exit: <>
4 sscanf: <>
5 printf: <>
6 mwert: <>
7 varianz: <>
```

was besagt, daß die Funktion `main()` vom Typ `int` ist und in Zeile 15 des Quelltextes `statistik.c` definiert wird. `main()` ruft seinerseits die Funktionen `puts`, `exit`, `sscanf`

und `printf` auf, die in `statistik.c` nicht definiert werden, da sie Teil der Standardbibliothek sind. Die Funktionen `mwert` und `varianz` werden ebenfalls aufgerufen und nicht definiert, da sie aus einer Privatbibliothek stammen.

Das Werkzeug `cxref(1)` erzeugt zu einer Gruppe von C-Quellfiles eine Kreuzreferenzliste aller Symbole, die nicht rein lokal sind. Der Aufruf

```
cxref fehler.c
```

gibt nach `stdout` eine Liste aus, deren erste Zeilen so aussehen:

```
fehler.c:

SYMBOL        FILE                  FUNCTION LINE

BUFSIZ        /usr/include/stdio.h  --       *10
EOF           /usr/include/stdio.h  --        70 *71
FILE          /usr/include/stdio.h  --       *18 78 123
                                              127 128 201
                                              223
FILENAME_MAX  /usr/include/stdio.h  --       *67
FOPEN_MAX     /usr/include/stdio.h  --       *68
L_ctermid     /usr/include/stdio.h  --       *193
L_cuserid     /usr/include/stdio.h  --       *194
L_tmpnam      /usr/include/stdio.h  --       *61
NULL          /usr/include/stdio.h  --        35 *36
PI            fehler.c              --       *27
P_tmpdir      /usr/include/stdio.h  --       *209
SEEK_CUR      /usr/include/stdio.h  --       *55
SEEK_END      /usr/include/stdio.h  --       *56
SEEK_SET      /usr/include/stdio.h  --        53 *54
TMP_MAX       /usr/include/stdio.h  --        63 *64
_CLASSIC_ANSI_TYPES /usr/include/stdio.h --   162
```

Durch die `include`-Datei `stdio.h` und gegebenenfalls durch Bibliotheksfunktionen geraten viele Namen in das Programm, von denen man nichts ahnt. Ferner gibt es einige Werkzeuge zur Ermittlung und Bearbeitung von Strings in Quellfiles und ausführbaren Programmen, teilweise beschränkt auf C-Programme (tt strings(1), `xstr(1)`).

Weitere wichtige Werkzeuge sind ein Lineal und Buntstifte, mit denen man zusammengehörende Namen oder Teile im Quelltext markiert.

1.2.8 Versionsverwaltung mit RCS, SCCS und CVS

Größere Projekte werden von zahlreichen, unter Umständen wechselnden Programmierern oder Autoren gemeinsam bearbeitet. In der Regel werden die so entstandenen Programmpakete über Jahre hinweg weiterentwickelt und vielleicht auf mehrere Systeme portiert. Die Arbeit vollzieht sich in mehreren **Stufen** parallel zur Zeitachse:

- Aufgabenstellung
- Aufgabenanalyse
- Umsetzung in eine Programmiersprache
- Testen
- Dokumentieren
- vorläufige Freigabe
- endgültige Freigabe
- Weiterentwicklung, Pflege

Des weiteren wird ein Programmpaket in viele überschaubare **Module** aufgeteilt. Von jedem Modul entstehen im Verlauf der Arbeit mehrere Fassungen oder **Versionen**. Der Zustand des ganzen Projektes läßt sich in einem dreidimensionalen Koordinatensystem mit den Achsen Modul, Stufe (Zeit) und Version darstellen. Das von WALTER F. TICHY entwickelte **Revision Control System RCS** ist ein Werkzeug, um bei dieser Entwicklung Ordnung zu halten. Es ist einfach handzuhaben und verträgt sich gut mit `make(1)`. Das RCS erledigt drei Aufgaben:

- Es führt Buch über die Änderungen an den Texten.
- Es ermöglicht, ältere Versionen wiederherzustellen, ohne daß diese vollständig gespeichert zu werden brauchen (Speichern von Differenzen).
- Es verhindert gleichzeitige schreibende Zugriffe mehrerer Benutzer auf denselben Text.

Sowie es um mehr als Wegwerfprogramme geht, sollte man `make(1)` und RCS einsetzen. Der geringe Aufwand zum Einarbeiten wird bei der weiteren Arbeit mehr als wett gemacht. Arbeiten mehrere Programmierer an einem Projekt, kommt man um RCS oder ähnliches nicht herum. Beide Werkzeuge sind auch für Manuskripte oder WWW-Dateien zu verwenden. RCS ist in den meisten Linux-Distributionen enthalten. Man beginnt folgendermaßen:

- Unterverzeichnis anlegen, hineinwechseln.
- Mit einem Editor die erste Fassung des Quelltextes schreiben. Irgendwo im Quelltext - z. B. im Kommentar - sollte `$Header: /home/debian/unix/unix8.tex,v 1.1.1.1 2005/02/06 20:01:11 wulf Exp $` oder `$Id: unix8.tex,v 1.1.1.1 2005/02/06 20:01:11 wulf Exp $` vorkommen, siehe unten. Dann übergibt man mit dem Kommando `ci filename` (check in) die Datei dem RCS. Dieses ergänzt die Datei durch Versionsinformationen und macht eine nur lesbare RCS-Datei (444) mit der Kennung `,v` daraus. Die ursprüngliche Datei löschen.
- Mit dem Kommando `co filename` (check out, ohne `,v`) bekommt man eine Kopie seiner Datei zurück, und zwar nur zum Lesen. Diese Kopie kann man mit allen UNIX-Werkzeugen bearbeiten, nur das Zurückschreiben mittels `ci` verweigert das RCS.
- Mit dem Kommando `co -l filename` wird eine les- und schreibbare Kopie erzeugt. Dabei wird die RCS-Datei für weitere, gleichzeitige Schreibzugriffe gesperrt (l = lock). Die Kopie kann man mit allen UNIX-Werkzeugen bearbeiten, Umbenennen wäre jedoch ein schlechter Einfall.
- Beim Zurückstellen mittels `ci filename` hat man Gelegenheit, einen kurzen Kommentar in die Versionsinformationen zu schreiben wie Grund und Umfang der Änderung. Mittels `rlog filename` werden die Versionsinformationen auf den Schirm geholt. Enthält der Quelltext die Zeichenfolge `$Log: unix8.tex,v $` – zweckmäßig im Kommentar am Anfang – so werden die Versionsinformationen auch dorthin übernommen. Dann hat man alles im Quellfile beisammen.
- Falls Sie sich mit `co -l filename` eine Kopie zum Editieren geholt und damit gleichzeitig das Original für weitere Schreibzugriffe gesperrt haben, anschließend die Kopie mit `rm(1)` löschen, so haben Sie nichts mehr zum Zurückstellen. In diesem Fall läßt sich die Sperre mit `rcs -u filename` aufheben. Besser ist es jedoch, auf die UNIX-Kommandos zu verzichten und nur mit den RCS-Kommandos zu arbeiten.

Das ist für den Anfang alles. Die RCS-Kommandos lassen sich in Makefiles verwenden. Die vom RCS vergebenen Zugriffsrechte können von UNIX-Kommandos (`chmod(1)`) überrannt werden, aber das ist nicht Sinn der Sache; der Einsatz von RCS setzt voraus, daß sich die Beteiligten diszipliniert verhalten.

Hier ein Makefile mit RCS-Kommandos für das nachstehende Sortierprogramm:

```
# makefile zu mysort.c, im RCS-System
# $Header: /home/debian/prog/quellen/sortmakef.tex,v 1.1.1.1 2005/02/06 19:44:00 wu

CC = /bin/cc
```

```
CFLAGS = -Aa -DDEBUG

all: mysort clean

mysort: mysort.o bubble.o
        $(CC) $(CFLAGS) -o mysort mysort.o bubble.o

mysort.o: mysort.c myheader.h
        $(CC) $(CFLAGS) -c mysort.c

bubble.o: bubble.c myheader.h
        $(CC) $(CFLAGS) -c bubble.c

mysort.c: mysort.c,v
        co mysort.c

bubble.c: bubble.c,v
        co bubble.c

myheader.h: myheader.h,v
        co myheader.h

clean:
        /bin/rm  -f *.c *.o *.h makefile
```

Quelle 1.12 : Makefile zum Sortierprogramm mysort.c

Da dieses Beispiel sich voraussichtlich zu einer kleinen Familie von Quelltexten ausweiten wird, legen wir eine privates include-Datei mit unseren eigenen, für alle Teile gültigen Werten an:

```
/* myheader.h zum Sortierprogramm, RCS-Beispiel
   W. Alex, Universitaet Karlsruhe, 04. Juli 1995
*/

/*
$Header: /home/debian/prog/quellen/sortinclude,v 1.1.1.1 2005/02/06 19:44:00 wulf Exp $
*/

int bubble(char *text);
int insert(char *text);

#define  USAGE  "Aufruf: mysort filename"
#define  NOTEXIST  "File existiert nicht"
#define  NOTREAD  "File ist nicht lesbar"
#define  NOTSORT  "Problem beim Sortieren"

#define  LINSIZ  64          /* Zeilenlaenge */
#define  MAXLIN  256         /* Anzahl Zeilen */
```

Quelle 1.13 : Include-Datei zum Sortierprogramm mysort.c

Nun das Hauptprogramm, das die Verantwortung trägt, aber sonst nicht viel tut. Hier ist der Platzhalter `$Header: /home/debian/unix/unix8.tex,v 1.1.1.1 2005/02/06 20:01:11 wulf Exp $` Bestandteil des Codes, die Versionsinformationen stehen also auch im ausführbaren Programm. Man könnte sogar mit ihnen etwas machen, ausgeben beispielsweise:

```
/* Sortierprogramm mysort, als Beispiel fuer RCS */

/*
$Log: mysort.tex,v $
Revision 1.1.1.1  2005/02/06 19:44:00  wulf
Anlegen des prog-Reps
```

```
*/

static char rcsid[] =
"$Header: /home/debian/prog/quellen/mysort.tex,v 1.1.1.1 2005/02/06 19:44:00 wulf E

#include <stdio.h>
#include "myheader.h"

int main(int argc, char *argv[])
{
long time1, time2;

/* Pruefung der Kommandozeile */

if (argc != 2) {
    puts(USAGE); return(-1);
}

/* Pruefung des Textfiles */

if (access(argv[1], 0)) {
    puts(NOTEXIST); return(-2);
}

if (access(argv[1], 4)) {
    puts(NOTREAD); return(-3);
}

/* Sortierfunktion und Zeitmessung */

time1 = time((long *)0);

if (bubble(argv[1])) {
    puts(NOTSORT); return(-4);
}

time2 = time((long *)0);

/* Ende */

printf("Das Sortieren dauerte %ld sec.\n", time2 - time1);
return 0;
}
```

Quelle 1.14 : C-Programm Sortieren, für RCS

Hier die Funktion zum Sortieren (Bubblesort, nicht optimiert). Der einzige Witz in dieser Funktion ist, daß wir nicht die Strings durch Umkopieren sortieren, sondern nur die Indizes der Strings. Ansonsten kann man hier noch einiges verbessern und vor allem auch andere Sortieralgorithmen nehmen. Man sollte auch das Einlesen und die Ausgabe vom Sortieren trennen:

```
/* Funktion bubble() (Bubblesort), als Beispiel fuer RCS
   W. Alex, Universitaet Karlsruhe, 04. Juli 1995 */

/*
   $Header: /home/debian/prog/quellen/bubble.tex,v 1.1.1.1 2005/02/06 19:44:00 wulf
*/

#include &#60;stdio.h&#62;
#include &#60;string.h&#62;
#include "myheader.h"

int bubble(char *text)
{
```

```
int i = 0, j = 0, flag = 0, z, line[MAXLIN];
char array[MAXLIN][LINSIZ];
FILE *fp;

#if DEBUG
printf("Bubblesort %s\n", text);
#endif

/* Einlesen */

if ((fp = fopen(text, "r")) == NULL) return(-1);

while ((!feof(fp)) && (i < MAXLIN)) {
    fgets(array[i++], LINSIZ, fp);
}

fclose(fp);

#if DEBUG
puts("Array:");
j = 0;
while (j < i) {
    printf("%s", array[j++]);
}
puts("Ende Array");
#endif

/* Sortieren (Bubblesort) */

for (j = 0; j < MAXLIN; j++)
    line[j] = j;

while (flag == 0) {
  flag = 1;
  for (j = 0; j < i; j++) {
    if (strcmp(array[line[j]], array[line[j + 1]]) > 0) {
         z = line[j + 1];
         line[j + 1] = line[j];
         line[j] = z;
         flag = 0;
    }
  }
}

/* Ausgeben nach stdout */

#if DEBUG
puts("Array:");
j = 0;
while (j < i) {
    printf("%d\n", line[j++]);
}
puts("Ende Array");
#endif

j = 0;
while (j < i) {
    printf("%s", array[line[j++]]);
}

/* Ende */

return 0;
}
```

Quelle 1.15 : C-Funktion Bubblesort

Bubblesort eignet sich für kleine Sortieraufgaben mit bis zu etwa hundert Elementen. Kopieren Sie sich die Bausteine in ein eigenes Verzeichnis und entwickeln Sie das Programm unter Verwendung des RCS weiter. Näheres siehe `rcsintro(5)`.

Anfangs erscheint das Arbeiten mit RCS bei kleinen Projekten als lästig, ähnlich wie das Anlegen eines Makefiles. Man gewöhnt sich aber schnell daran und spart sofort das Eintragen des Änderungsdatums von Hand. Nach kurzer Zeit ist man für die selbst auferlegte Ordnung dankbar.

Das **Source Code Control System SCCS** verwaltet die Versionen der Module, indem es die erste Fassung vollständig speichert und dann jeweils die Differenzen zur nächsten Version, während RCS die jüngste Version speichert und die älteren aus den Differenzen rekonstruiert.

Alle Versionen eines Programmes samt den Verwaltungsdaten werden in einer einzigen SCCS-Datei namens `s.filename` abgelegt, auf das schreibend nur über besondere SCCS-Kommandos zugegriffen werden kann. Das erste dieser Kommandos ist `admin(1)` und erzeugt aus einem C-Quellfile `program.c` das zugehörige SCCS-Dokument:

```
admin -iprogram.c s.program.c
```

Mit `admin(1)` lassen sich noch weitere Aufgaben erledigen, siehe Referenz-Handbuch. Mittels `get(1)` holt man das Quellfile wieder aus dem SCCS-Dokument heraus, mitttels `delta(1)` gibt man eine geänderte Fassung des Quellfiles an das SCCS-Dokument zurück.

RCS und SCCS arbeiten auf Datei-Ebene. Bei größeren Projekten ist es wünschenswert, mehrere Dateien gemeinsam oder ganze Verzeichnisse in die Versionsverwaltung einzubeziehen. Dies leistet das **Concurrent Versions System** (CVS). Es baut auf RCS auf und erweitert dessen Funktionalität außerdem um eine Client-Server-Architektur. Die beteiligten Dateien und Verzeichnisse können auf verschiedenen Computern im Netz liegen. Im Gegensatz zu RCS, das zu einem Zeitpunkt immer nur einem Benutzer das Schreiben gestattet, verfolgt CVS eine sogenannte optimistische Kooperationsstrategie. Mehrere Programmierer können gleichzeitig auf Kopien derselben Version (Revision) arbeiten. Beim Zurückschreiben wird ein Abgleich mit der in der zentralen Versionsbibliothek (Repository) abgelegten Fassung erzwungen, um zu verhindern, daß parallel durchgeführte und bereits zurückgeschriebene Versionen überschrieben werden. Diese Strategie kann zu Konflikten führen, die per Hand aufgelöst werden müssen. Während das Einrichten eines CVS-Projektes Überblick erfordert, ist das Arbeiten unter CVS nicht schwieriger als unter RCS. Einzelheiten wie so oft am einfachsten aus dem Netz, wo außer dem Programmpaket selbst auch kurze oder ausführliche, deutsche oder englische Anleitungen zu finden sind. Unter den Namen *WinCVS* und *MacCVS* liegen Fassungen für weitere Betriebssysteme im Netz.

Der Oberbegriff des ganzen Gebietes lautet *Software Configuration Management* (SCM) oder allgemeiner *Configuration Management* (CM). Lassen Sie einmal eine Suchmaschine darauf los, es gibt mehrere freie oder kommerzielle Produkte sowie Übersichten, Einführungen und Tutorials dazu.

Ist die Entwicklung einer Software oder eines Manuskriptes vorläufig abgeschlossen, geht es an die Pflege. Dazu gehört unter anderem das Management der im Betrieb der Software auftauchenden Probleme. Auch hierfür gibt es Werkzeuge, beispielsweise `gnats` aus dem GNU-Projekt. Aber das sprengt den Rahmen dieses Buches.

CASE bedeutet *Computer Aided Software Engineering*. An sich ist das nichts Neues, beim Programmieren hat man schon immer Computer eingesetzt. Das Neue bei CASE Tools wie SoftBench von Hewlett-Packard besteht darin, daß die einzelnen Programmierwerkzeuge wie syntaxgesteuerte Editoren, Compiler, Linker, Builder (`make(1)`), Analysewerkzeuge, Debugger, Versionskontrollsysteme sowie die Dokumentation unter einer einheitlichen, heutzutage grafischen Oberfläche – hier das X Window System und Motif - zusammengefaßt werden. Allgemein heißt das Ganze Programmier- oder Entwicklungsumgebung (Integrated development environment, IDE). Damit zu arbeiten ist die moderne Form des Programmierens und kann effektiv sein.

1.2.9 Systemaufrufe

1.2.9.1 Was sind Systemaufrufe?

Dem Programmierer stehen zwei Hilfsmittel[17] zur Verfügung, um seine Wünsche auszudrücken:

- die Schlüsselwörter (Wortsymbole) der Programmiersprache,
- die Systemaufrufe des Betriebssystems.

Die **Schlüsselwörter** (keyword, mot-clé) der Programmiersprache (C/C++, FORTRAN oder PASCAL) sind auch unter verschiedenen Betriebssystemen (PC-DOS, OS/2 oder UNIX) dieselben. Sie gehören zur Programmiersprache, das heißt zum Compiler. Die **Systemaufrufe** (system call, system primitive, fonction système) eines Betriebssystems (UNIX) sind für alle Programmiersprachen (C, FORTRAN, PASCAL, COBOL) dieselben. Sie gehören zum Betriebssystem. Man findet auch die Bezeichnung Kernschnittstellenfunktion, die besagt, dass ein solcher Aufruf sich unmittelbar an den Kern des Betriebssystems richtet. Der Kreis der Systemaufrufe liegt fest und kann nicht ohne Eingriffe in den Kern des Betriebssystems verändert werden. Da UNIX zum großen Teil in C geschrieben ist, sind die Systemaufrufe von UNIX C-Funktionen, die sich in ihrer Syntax nicht von eigenen oder fremden C-Funktionen unterscheiden. Deshalb müssen auch FORTRAN- oder PASCAL-Programmierer etwas von der Programmiersprache C verstehen. Im Handbuch werden die Systemaufrufe in Sektion (2) beschrieben.

Bei POSIX-konformen Betriebssystemen spricht man statt von Systemaufrufen besser von POSIX-Funktionen, da der POSIX-Standard offen lässt, ob diese vom Betriebssystem zur Verfügung gestellten Funktionen als Systemaufrufe oder als Bibliothek verwirklicht sind. Auf jeden Fall gehören sie zum Betriebssystem, nicht zum Compiler. Die Unterscheidung spielt eine Rolle, wenn man für verschiedene Betriebssysteme und/oder Compiler programmiert. Der Programmierer muss wissen, woher seine Funktionen stammen.

In Sektion (3) finden sich vorgefertigte **Unterprogramme**, **Subroutinen** oder **Standardfunktionen** (standard function, fonction élémentaire) für häufig vorkommende Aufgaben. Für den Anwender besteht kein Unterschied zu den Systemaufrufen. Streng genommen gehören diese Standardfunktionen jedoch zu den jeweiligen Programmiersprachen (zum Compiler) und nicht zum Betriebssystem. Der Kreis der Standardfunktionen ist beliebig ergänzbar. Um den Benutzer zu verwirren, sind die Systemaufrufe und die Standardfunktionen in *einer* Funktionsbibliothek (`/lib/libc.a` und andere) vereinigt.

Die Aufgabenverteilung zwischen Schlüsselwörtern, Systemaufrufen und Standardfunktionen ist in gewissem Umfang willkürlich. Systemaufrufe erledigen Aufgaben, die aus dem Aufbau und den kennzeichnenden Eigenschaften des Betriebssystems herrühren, bei UNIX also in erster Linie

- Ein- und Ausgabe auf unterster Stufe,
- Umgang mit Prozessen,
- Umgang mit dem Datei-System,
- Sicherheitsvorkehrungen.

Das Öffnen einer Datei zum Lesen oder Schreiben ist Sache eines Systemaufrufs (`open(2)`), Sortieren hingegen Sache einer Standardfunktion (`qsort(3)`). Es gibt aber zusätzlich auch Standardfunktionen zum Umgang mit Dateien, die den jeweiligen Systemaufruf komfortabel verpacken (`fopen(3)`). Nach außen definiert die Menge der Systemaufrufe das Betriebssystem. Zwei Systeme, die in ihren Aufrufen übereinstimmen, sind für den Benutzer identisch. Neue Funktionalitäten des Betriebssystems stellen sich dem Programmierer als neue Systemaufrufe dar, siehe zum Beispiel unter `stream(2)`.

[17] Standardfunktionen sind erst verfügbar, nachdem andere Programmierer sie geschrieben haben.

Einige UNIX-Systemaufrufe haben gleiche oder ähnliche Aufgaben wie Shell-Kommandos. Wenn man die Zeit wissen möchte, verwendet man im Dialog das Shell-Kommando `date(1)`. Will man diese Information aus einem eigenen Programm heraus abfragen, greift man auf den Systemaufruf `time(2)`[18] zurück. Das Shell-Kommando ist ein in ein C-Programm verpackter Systemaufruf.

In Linux/UNIX sind Systemaufrufe **Funktionen** der Programmiersprache C. Eine Funktion übernimmt beim Aufruf Argumente oder Parameter und gibt ein Ergebnis zurück. Dieser Mechanismus wird **Parameterübergabe** genannt. Man muss ihn verstanden haben, um Funktionen in eigenen Programmen verwenden zu können. Eine Erklärung findet sich im Skriptum *Programmieren in C/C++*.

1.2.9.2 Beispiel Systemzeit (time)

Im folgenden Beispiel wird der Systemaufruf `time(2)` verwendet. `time(2)` liefert die Zeit in Sekunden seit 00:00:00 Greenwich Mean Time, 1. Januar 1970. Computeruhren laufen übrigens erstaunlich ungenau, falls sie nicht durch eine Funkuhr oder über das Netz synchronisiert werden. Ferner brauchen wir die Standardfunktion `gmtime(3)`, Beschreibung unter `ctime(3)`, die aus den obigen Sekunden eine Struktur erzeugt, die Datum und Uhrzeit enthält. Die Umrechnung von Greenwich auf Karlsruhe nehmen wir selbst vor. Eleganter wäre ein Rückgriff auf die Zeitzonen-Variable der Umgebung. Laut Referenz-Handbuch hat `time(2)` die Syntax

```
long time ((long *) 0)
```

Die Funktion verlangt ein Argument vom Typ Pointer auf long integer, und zwar im einfachsten Fall den Nullpointer. Der Returnwert ist vom Typ long integer. Der größte Wert dieses Typs liegt etwas über 2 Milliarden. Damit läuft diese Uhr etwa 70 Jahre. Die Subroutine `gmtime(3)` hat die Syntax

```
#include <time.h>
struct tm *gmtime(clock)
long *clock
```

Die Funktion `gmtime(3)` verlangt ein Argument `clock` vom Typ Pointer auf long integer. Wir müssen also den Returnwert von `time(2)` in einen Pointer umwandeln (referenzieren). Der Rückgabewert der Funktion `gmtime(3)` ist ein Pointer auf eine Struktur namens `tm`. Diese Struktur ist in der include-Datei `time.h` definiert. Die include-Dateien sind lesbarer Text; es ist ratsam hineinzuschauen. In der weiteren Beschreibung zu `ctime(3)` wird die Struktur `tm` erläutert:

```
struct tm {
       int tm_sec;        /* seconds (0 - 59) */
       int tm_min;        /* minutes (0 - 59) */
       int tm_hour;       /* hours (0 - 23) */
       int tm_mday;       /* day of month (1 - 31) */
       int tm_mon;        /* month of year (0 - 11) */
       int tm_year;       /* year - 1900 */
       int tm_wday;       /* day of week (sunday = 0) */
       int tm_yday;       /* day of year (0 - 365) */
       int tm_isdst;      /* daylight saving time */
}
```

Von den beiden letzten Komponenten der Struktur machen wir keinen Gebrauch. Da die Komponenten alle vom selben Typ sind, ist statt der Struktur auch ein Array denkbar. Vermutlich wollte sich der Programmierer den Weg offenhalten, künftig auch andere Typen aufzunehmen (Zeitzone). Das Programm, das die Quelle zu dem Kommando `zeit` aus der ersten Übung ist, sieht folgendermaßen aus:

[18] In HP-UX. In ANSI-C ist eine Standardfunktion `time(3)` enthalten.

```
/* Ausgabe der Zeit auf Bildschirm */
/* Compileraufruf cc -o zeit zeit.c */

#include <stdio.h>
#include <time.h>

char *ptag[] = {"Sonntag,    ", "Montag,     ",
                "Dienstag,   ", "Mittwoch,   ",
                "Donnerstag,", "Freitag,    ",
                "Samstag,    "};
char *pmon[] = {"Januar", "Februar", "Maerz", "April",
                "Mai", "Juni", "Juli", "August",
                "September", "Oktober", "November",
                "Dezember"};

main()
{
long sec, time();
struct tm *gmtime(), *p;

sec = time((long *) 0) + 3600;   /* MEZ = GMT + 3600 */
p = gmtime(&sec);
printf("%s %d. ", ptag[p->tm_wday], p->tm_mday);
printf("%s %d     ", pmon[p->tm_mon], p->tm_year +1900);
printf("%d:%02d MEZ\n", p->tm_hour, p->tm_min);
}
```

Quelle 1.16 : C-Programm zur Anzeige der Systemzeit

Nun wollen wir dieselbe Aufgabe mit einem FORTRAN-Programm bewältigen. Der UNIX-Systemaufruf `time(2)` bleibt, für die C-Standardfunktion `gmtime(3)` suchen wir die entsprechende FORTRAN-Routine. Da wir keine finden, müssen wir sie entweder selbst schreiben (was der erfahrene Programmierer scheut) oder nach einem Weg suchen, eine beliebige C-Standardfunktion in ein FORTRAN-Programm hineinzuquetschen.

Der Systemaufruf `time(2)` macht keinen Kummer. Er benötigt ein Argument vom Typ Pointer auf long integer, was es in FORTRAN gibt. Der Rückgabewert ist vom Typ long integer, auch kein Problem. Die C-Standardfunktion `gmtime(3)` erwartet ein Argument vom Typ Pointer auf long integer, was machbar wäre, aber ihr Ergebnis ist ein Pointer auf eine Struktur. Das hat FORTRAN noch nie gesehen[19]. Deshalb weichen wir auf die C-Standardfunktion `ctime(3)` aus, deren Rückgabewert vom Typ Pointer auf character ist, was es in FORTRAN näherungsweise gibt. In FORTRAN ist ein Zeichen ein String der Länge eins. Strings werden per Deskriptor übergeben. Ein **String-Deskriptor** ist der Pointer auf das erste Zeichen *und* die Anzahl der Zeichen im String als Integerwert. Das Programm sieht dann so aus:

```
      program zeit

$ALIAS foratime = 'sprintf' c

      integer*4 time, tloc, sec, ctime
      character atime*26

      sec = time(tloc)

      call foratime(atime, '%s'//char(0), ctime(sec))
      write(6, '(a)') atime

      end
```

Quelle 1.17 : FORTRAN-Programm zur Anzeige der Systemzeit

Die **ALIAS-Anweisung** ist als Erweiterung zu FORTRAN 77 in vielen Compilern ent-

[19]FORTRAN 90 kennt Strukturen.

halten und dient dazu, den Aufruf von Unterprogrammen anderer Sprachen zu ermöglichen. Der Compiler weiß damit, dass das Unterprogramm außerhalb des Programms – zum Beispiel in einer Bibliothek – einen anderen Namen hat als innerhalb des Programms. Wird eine Sprache angegeben (hier C), so erfolgt die Parameterübergabe gemäß der Syntax dieser Sprache. Einzelheiten siehe im Falle unserer Anlage im HP FORTRAN 77/HP-UX Reference Manual im Abschnitt *Compiler Directives*.

Die Anweisung teilt dem Compiler mit, dass hinter der FORTRAN-Subroutine `foratime` die C-Standard-Funktion `sprintf(3)` steckt und dass diese nach den Regeln von C behandelt werden soll. Der Rückgabewert von `sprintf(3)` (die Anzahl der ausgegebenen Zeichen) wird nicht verwertet, deshalb ist `foratime` eine FORTRAN-Subroutine (keine Funktion), die im Programm mit `call` aufgerufen werden muss.

Der Systemaufruf `time(2)` verlangt als Argument einen Pointer auf `long integer`, daher ist `tloc` als vier Bytes lange Integerzahl deklariert. `tloc` spielt weiter keine Rolle. Die Übergabe als Pointer (by reference) ist in FORTRAN Standard für Zahlenvariable und braucht nicht eigens vereinbart zu werden. Der Rückgabewert von `time` geht in die Variable `sec` vom Typ `long integer` = `integer*4`.

Die `call`-Zeile ruft die Subroutine `foratime` alias C-Funktion `sprintf(3)` auf. Diese C-Funktion erwartet drei Argumente: den Ausgabestring als Pointer auf `char`, einen Formatstring als Pointer auf `char` und die auszugebende Variable von einem Typ, wie er durch den Formatstring bezeichnet wird. Der Rückgabewert der Funktion `ctime(3)` ist ein Pointer auf `char`. Da dies kein in FORTRAN zulässiger Typ ist, deklarieren wir die Funktion ersatzweise als vom Typ 4-Byte-integer. Der Pointer lässt sich auf jeden Fall in den vier Bytes unterbringen. Nach unserer Erfahrung reichen auch zwei Bytes, ebenso funktioniert der Typ `logical`, nicht jedoch `real`.

Der Formatstring besteht aus der Stringkonstanten `%s`, gefolgt von dem ASCII-Zeichen Nr. 0, wie es bei Strings in C Brauch ist. Für `sprintf(3)` besagt dieser Formatstring, das dritte Argument – den Rückgabewert von `ctime(3)` – als einen String aufzufassen, das heißt als Pointer auf das erste Element eines Arrays of characters.

`atime` ist ein FORTRAN-String-Deskriptor, dessen erste Komponente ein Pointer auf character ist. Damit weiß `sprintf(3)`, wohin mit der Ausgabe. Die `write`-Zeile ist wieder pures FORTRAN.

An diesem Beispiel erkennen Sie, dass Sie auch als FORTRAN- oder PASCAL-Programmierer etwas von C verstehen müssen, um die Systemaufrufe und C-Standardfunktionen syntaktisch richtig zu gebrauchen.

Bei manchen FORTRAN-Compilern (Hewlett-Packard, Microsoft) lassen sich durch einen einfachen **Interface-Aufruf** Routinen fremder Sprachen so verpacken, dass man sie übernehmen kann, ohne sich um Einzelheiten kümmern zu müssen.

1.2.9.3 Beispiel Datei-Informationen (access, stat, open, close)

In einem weiteren Beispiel wollen wir mithilfe von Systemaufrufen Informationen über eine Fatei gewinnen, dazu noch eine Angabe aus der Sitzungsumgebung. Die Teile des Programms lassen sich einfach in andere C-Programme übernehmen.

Dieses Programm soll beim Aufruf (zur Laufzeit, in der Kommandozeile) den Namen der Datei als Argument übernehmen, wie wir es von UNIX-Kommandos her kennen. Dazu ist ein bestimmter Formalismus vorgesehen:

```
int main(int argc, char *argv[], char *envp[])
```

Die Funktion `main()` übernimmt die Argumente `argc`, `argv` und gegebenenfalls `envp`. Das Argument `argc` ist der **Argument Counter**, eine Ganzzahl. Sie ist gleich der Anzahl der Argumente in der Kommandozeile beim Aufruf des Programms. Das Kommando selbst ist das erste Argument, also hat `argc` mindestens den Wert 1. Das Argument `argv` ist der **Argument Vector**, ein Array of Strings, also ein Array of Arrays of Characters. Der erste String, Index 0, ist das Kommando; die weiteren Strings sind die mit dem Kommando übergebenen Argumente, hier der Name der gefragten Datei. Der **Environment Pointer** `envp` wird nur benötigt, falls man Werte aus der Umgebung abfragt. Es ist wie `argv` ein Array of Strings.

Die Namen `argc`, `argv` und `envp` sind willkürlich, aber üblich. Typ und Reihenfolge sind vorgegeben.

Die Umgebung besteht aus Strings (mit Kommando `set (Shell)` anschauen). In der `for`-Schleife werden die Strings nacheinander mittels der Funktion `strncmp(3)` (siehe `string(3)`) mit dem String LOGNAME verglichen. Das Ergebnis ist der Index `i` des gesuchten Strings im Array `envp[]`.

Den Systemaufruf `access(2)` finden wir in der Sektion (2) des Referenz-Handbuches. Er untersucht die Zugriffsmöglichkeiten auf eine Datei und hat die Syntax:

```
int access(char *path, int mode)
```

Der Systemaufruf erwartet als erstes Argument einen String, nämlich den Namen der Datei. Wir werden hierfür `argv[1]` einsetzen. Als zweites steht eine Ganzzahl, die die Art des gefragten Zugriffs kennzeichnet. Falls der gefragte Zugriff möglich ist, liefert `access(2)` den Wert null zurück, der in einem C-Programm zugleich die Bedeutung von logisch falsch (FALSE) hat und deshalb in den `if`-Zeilen negiert wird.

Den Systemaufruf `stat(2)` finden wir ebenfalls in Sektion 2. Er ermittelt Dateiinformationen aus der **Inode** und hat die Syntax

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(path, buf)
char *path;
struct stat *buf;
```

Sein erstes Argument ist wieder der Dateiname, das zweite der Name eines Puffers zur Aufnahme einer Struktur, die die Informationen enthält. Diese Struktur vom Typ `stat` ist in der include-Datei `/usr/include/sys/stat.h` deklariert, das seinerseits Bezug nimmt auf Deklarationen in `/usr/include/types.h`. Auch einige Informationen wie `S_IFREG` sind in `sys/stat.h` definiert. Die Zeitangaben werden wie im vorigen Abschnitt umgerechnet.

In UNIX-Datei-Systemen enthält jede Datei am Anfang eine **Magic Number**, die über die Art der Datei Auskunft gibt (`man magic`). Mittels des Systemaufrufs `open(2)` wird die fragliche Datei zum Lesen geöffnet, mittels `lseek(2)` der Lesezeiger auf die Magic Number gesetzt und mittels `read(2)` die Zahl gelesen. Der Systemaufruf `close(2)` schließt die Datei wieder. Die Systemaufrufe findet man unter ihren Namen in Sektion (2), eine Erläuterung der Magic Numbers unter `magic(4)`. Nun das Programm:

```
/* Informationen ueber eine Datei */

#define  MEZ 3600

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <fcntl.h>
#include <magic.h>

void exit(); long lseek();

int main(argc, argv, envp)

    int argc; char *argv[], *envp[];

{

int i, fildes;
struct stat buffer;
long asec, msec, csec;
struct tm *pa, *pm, *pc;
```

```
if (argc < 2) {
    puts("Dateiname fehlt"); return (-1);
}

/* Informationen aus dem Environment */

for (i = 0; envp[i] != NULL; i++)
    if (!(strncmp(envp[i], "LOGNAME", 4)))
        printf("\n%s\n", envp[i]);

/* Informationen mittels Systemaufruf access(2) */

printf("\nFile heisst: %8s\n", argv[1]);

if (!access(argv[1], 0))
    puts("File existiert");
else
    puts("File existiert nicht");

if (!access(argv[1], 1))
    puts("File darf ausgefuehrt werden");
else
    puts("File darf nicht ausgefuehrt werden");

if (!access(argv[1], 2))
    puts("File darf beschrieben werden");
else
    puts("File darf nicht beschrieben werden");

if (!access(argv[1], 4))
    puts("File darf gelesen werden");
else
    puts("File darf nicht gelesen werden");

/* Informationen aus der Inode, Systemaufruf stat(2) */

if (!(stat(argv[1], &buffer))) {
    printf("\nDevice:          %ld\n", buffer.st_dev);
    printf("Inode-Nr.:       %lu\n", buffer.st_ino);
    printf("File Mode:       %hu\n\n", buffer.st_mode);

    switch(buffer.st_mode & S_IFMT) {
        case S_IFREG:
            {
            puts("File ist regulaer");
            break;
            }
        case S_IFDIR:
            {
            puts("File ist ein Verzeichnis");
            break;
            }
        case S_IFCHR:
        case S_IFBLK:
        case S_IFNWK:
            {
            puts("File ist ein Special File");
            break;
            }
        case S_IFIFO:
            {
            puts("File ist eine Pipe");
            break;
            }
        default:
            {
```

```
            puts("Filetyp unbekannt (Inode)");
            }
    }
    printf("\nLinks:           %hd\n", buffer.st_nlink);
    printf("Owner-ID:        %hu\n", buffer.st_uid);
    printf("Group-Id:        %hu\n", buffer.st_gid);
    printf("Device-ID:       %ld\n", buffer.st_rdev);
    printf("Filegroesse:     %ld\n", buffer.st_size);

    asec = buffer.st_atime + MEZ; pa = gmtime(&asec);
    msec = buffer.st_mtime + MEZ; pm = gmtime(&msec);
    csec = buffer.st_ctime + MEZ; pc = gmtime(&csec);

    printf("Letzter Zugriff: %d. %d. %d\n",
            pa->tm_mday, pa->tm_mon + 1, pa->tm_year);
    printf("Letzte Modifik.: %d. %d. %d\n",
            pm->tm_mday, pm->tm_mon + 1, pm->tm_year);
    printf("Letzte Stat.Ae.: %d. %d. %d\n",
            pc->tm_mday, pc->tm_mon + 1, pc->tm_year);

}
else
    puts("Kein Zugriff auf Inode");

/* Pruefung auf Text oder Code (magic number) */
/* Systemaufrufe open(2), lseek(2), read(2), close(2) */
/* Magic Numbers siehe magic(4) */

{
    MAGIC   magbuf;

    fildes = open(argv[1], O_RDONLY);
    if (lseek(fildes, MAGIC_OFFSET, 0) >= (long)0) {
        read(fildes, &magbuf, sizeof magbuf);
        switch(magbuf.file_type) {
            case RELOC_MAGIC:
                {
                puts("File ist relocatable");
                break;
                }
                case EXEC_MAGIC:
                case SHARE_MAGIC:
                case DEMAND_MAGIC:
                {
                puts("File ist executable");
                break;
                }
                case DL_MAGIC:
                case SHL_MAGIC:
                {
                puts("File ist Library");
                break;
                }
            default:
                puts("Filetyp unbekannt (Magic Number)");
                lseek(fildes, 0L, 0);
        }
    }
    else {
        puts("Probleme mit dem Filepointer");
    }

}
close(fildes);
}
```

Quelle 1.18 : C-Programm zum Abfragen von Informationen über eine Datei

Die Verwendung von Systemaufrufen oder Standardfunktionen in C-Programmen ist nicht schwieriger als der Gebrauch anderer Funktionen. Man muss sich nur an die im Referenz-Handbuch Sektionen (2) und (3) nachzulesende Syntax halten. Es empfiehlt sich, die genannten Sektionen einmal durchzublättern, um eine Vorstellung davon zu gewinnen, wofür es Systemaufrufe und Standardfunktionen gibt. Die Ausgabe des Programms sieht folgendermaßen aus:

```
LOGNAME=wualex1

Datei heisst:        a.out
Datei existiert
Datei darf ausgefuehrt werden.
Datei darf nicht beschrieben werden.
Datei darf gelesen werden.

Device:             13
Inode-Nr.:          43787
File Mode:          33216

Datei ist regulaer

Links:              1
Owner-ID:           101
Group-ID:           20
Device-ID:          102536
Dateigroesse:        53248
Letzter Zugriff:    24. 1. 91
Letzte Modifik.:    24. 1. 91
Letzte Stat.Ae.:    24. 1. 91
Datei ist executable
```

Die Bedeutung von `File Mode` finden Sie bei `mknod(2)`. Es handelt sich um ausführliche Informationen über die Zugriffsrechte usw. Ähnliche Auskünfte über eine Datei liefert das Kommando `chatr(1)`.

1.2.9.4 Beispiel Prozesserzeugung (exec, fork)

Zunächst ein kleines, aber fieses Programm namens `forkbomb`, mit dem man die Robustheit seines Systems auf die Probe stellen kann.

```
(kommt demnaechst)
```

Quelle 1.19 : C-Programm zum Erzeugen vieler Prozesse (Fork-Bombe)

Der Systemaufruf `fork(2)` erzeugt eine Kopie des aufrufenden Prozesses mit einer neuen Prozess-ID. Im Beispiel wird `fork(2)` in einer ewigen `for`-Schleife aufgerufen.

1.2.10 Begriffe Programmer's Workbench

Folgende Begriffe sollten klarer geworden sein:

- Archiv, Bibliothek
- Builder (`make`)
- Debugger
- Linker

- Profiler
- Quelltext oder -code, ausführbarer Code
- Übersetzer, Compiler, Interpreter
- Systemaufruf – Standardfunktion
- Versionsverwaltung

Folgende Kommandos sollten beherrscht werden:

- `cc` oder `gcc`
- `lint`
- Anfänge von `make`
- Anfänge von RCS (`ci`, `co`)

1.2.11 Memo Programmer's Workbench

- Die Programmquellen werden mit einem Editor geschrieben.
- Mit dem Syntaxprüfer `lint(1)` läßt sich die syntaktische Richtigkeit von C-Programmen prüfen, leider nicht die von C++-Programmen.
- Schon bei kleinen Programmierprojekten ist das Werkzeug `make(1)` dringend zu empfehlen. Der Compileraufruf vereinfacht sich wesentlich. Auch für Texte verwendbar.
- Mit einem Compiler wird der Quellcode in den Maschinencode des jeweiligen Prozessors übersetzt.
- Der schwerste Hammer bei der Fehlersuche ist ein Debugger, lernbedürftig, aber nicht immer vermeidbar.
- Programmfunktionen (aber auch andere Dateien) lassen sich in Bibliotheken archivieren, die bequemer zu handhaben sind als eine Menge von einzelnen Funktionen.
- Bei größeren Projekten kommt man nicht um ein Kontrollsystem wie RCS oder CVS herum, vor allem dann, wenn mehrere Personen beteiligt sind. Das Lernen kostet Zeit, die aber beim Ringen mit dem Chaos mehr als wettgemacht wird.
- CASE-Tools vereinigen die einzelnen Werkzeuge unter einer gemeinsamen Benutzeroberfläche. Der Programmierer braucht gar nicht mehr zu wissen, was ein Compiler ist.
- Systemaufrufe sind die Verbindungen des Betriebssystems nach oben, zu den Anwendungsprogrammen hin. Sie sind Teil des Betriebssystems.
- Systemaufrufe haben vorwiegend mit Prozessen, den Datei-Systemen und der Ein- und Ausgabe zu tun.
- UNIX-Systemaufrufe sind C-Funktionen, die sich im Gebrauch nicht von anderen C-Funktionen unterscheiden.
- C-Standardfunktionen gehören zum C-Compiler, nicht zum Betriebssystem.
- Ein FORTRAN-Programmierer auf einem UNIX-System ist auf die UNIX-Systemaufrufe angewiesen, nicht aber auf die C-Standardfunktionen (dafür gibt es FORTRAN-Standardfunktionen). Dasselbe gilt für jede andere Programmiersprache.

1.2.12 Übung Programmer's Workbench

Anmelden wie gewohnt. Zum Üben brauchen wir ein kleines Programm mit bestimmten Fehlern. Legen Sie mit `mkdir prog` ein Unterverzeichnis `prog` an, wechseln Sie mit `cd prog` dorthin und geben Sie mit `vi fehler.c` folgendes C-Programm (ohne den Kommentar) unter dem Namen `fehler.c` ein:

```
/* Uebungsprogramm mit mehreren Fehlern */

/* 1. Fehler: Es wird eine symbolische Konstante PI
   definiert, die nicht gebraucht wird. Dieser Fehler
   hat keine Auswirkungen und wird von keinem
   Programm bemerkt.
   2. Fehler: Eine Ganzzahl-Variable d wird deklariert,
   aber nicht gebraucht. Dieser Fehler hat keine
   Auswirkungen, wird aber von lint beanstandet.
   3. Fehler: Die Funktion scanf verlangt Pointer als
   Argument, es muss &a heissen. Heimtueckischer
   Syntaxfehler. lint gibt eine irrefuehrende Warnung
   aus, der Compiler merkt nichts. Zur Laufzeit ein
   memory fault.
   4. Fehler: Es wird durch nichts verhindert, dass fuer
   b eine Null eingegeben wird. Das kann zu einem
   Laufzeitfehler fuehren, wird weder von lint noch
   vom Compiler bemerkt.
   5. Fehler: Es sollte die Summe ausgerechnet werden,
   nicht der Quotient. Logischer Fehler, wird weder
   von lint noch vom Compiler bemerkt.
   6. Fehler: Abschliessende Klammer fehlt. Syntaxfehler,
   wird von lint und Compiler beanstandet.

   Darueberhinaus spricht lint noch Hinweise bezueglich
   main, printf und scanf aus. Diese Funktionen sind
   aber in Ordnung, Warnungen ueberhoeren.  */

#define PI 3.14159
#include <stdio.h>

int main()
{

    int a, b, c, d;

    puts("Bitte 1. Summanden eingeben: ");
    scanf("%d", a);
    puts("Bitte 2. Summanden eingeben: ");
    scanf("%d", &b);
    c = a / b;
    printf("Die Summe ist: %d\n", c);
```

Quelle 1.20 : C-Programm mit Fehlern

Als erstes lassen wir den Syntaxprüfer `lint(1)` auf das Programm los:

```
lint fehler.c
```

und erhalten das Ergebnis:

```
fehler.c
==============
(36)  warning: a may be used before set
(41)  syntax error
(41)  warning: main() returns random value to environment
```

```
==============
function returns value which is always ignored
    printf     scanf
```

Zeile 41 ist das Programmende, dort steckt ein Fehler. Die Warnungen sind nicht so dringend. Mit dem `vi(1)` ergänzen wir die fehlende geschweifte Klammer am Schluß. Der Fehler hätte uns eigentlich nicht unterlaufen dürfen, da der `vi(1)` eine Hilfe zur Klammerprüfung bietet (Prozentzeichen). Neuer Lauf von `lint(1)`:

```
fehler.c
==============
(36)  warning: a may be used before set
(33)  warning: d unused in function main
(41)  warning: main() returns random value to environment

==============
function returns value which is always ignored
    printf     scanf
```

Wir werfen die überflüssige Variable d in der Deklaration heraus. Nochmals `lint(1)`.

```
fehler.c
==============
(36)  warning: a may be used before set
(41)  warning: main() returns random value to environment

==============
function returns value which is always ignored
    printf     scanf
```

Jetzt ignorieren wir die Warnung von `lint(1)` bezüglich der Variablen a (obwohl heimtückischer Fehler, aber das ahnen wir noch nicht). Wir lassen kompilieren und rufen das kompilierte Programm `a.out(4)` auf:

```
cc fehler.c
a.out
```

Der Compiler hat nichts zu beanstanden. Ersten Summanden eingeben, Antwort: `memory fault` oder `Bus error - core dumped`. Debugger[20] einsetzen, dazu nochmals mit der Option -g und dem vom Debugger verwendeten Objektfile `/usr/lib/xdbend.o` kompilieren und anschließend laufen lassen, um einen aktuellen Speicherauszug (Coredump) zu erzeugen:

```
cc -g fehler.c /usr/lib/xdbend.o
chmod 700 a.out
a.out
xdb
```

Standardmäßig greift der Debugger auf die ausführbare Datei `a.out(4)` und das beim Zusammenbruch erzeugte Corefile `core(4)` zurück. Er promptet mit >. Wir wählen mit der Eingabe `s` Einzelschritt-Ausführung. Mehrmals mit RETURN weitergehen, bis Aufforderung zur Eingabe von a kommt (kein Prompt). Irgendeinen Wert für a eingeben. Fehlermeldung des Debuggers `Bus error`. Wir holen uns weitere Informationen vom Debugger:

```
T   (stack viewing)
s   (Einzelschritt)
q   (quit)
```

[20]Real programmers can read core dumps.

Nachdem wir wissen, daß der Fehler nach der Eingabe von a auftritt, schauen wir uns die Zeile mit `scanf( ..., a)` an und bemerken, daß wir der Funktion `scanf(3)` eine Variable statt eines Pointers übergeben haben (`man scanf` oder im Anhang nachlesen). Wir ersetzen also `a` durch `&a`. Das Compilieren erleichtern wir uns durch `make(1)`. Wir schreiben eine Datei namens `makefile` mit folgenden Zeilen:

```
fehler: fehler.c
    cc fehler.c -o fehler
```

und rufen anschließend nur noch das Kommando `make(1)` ohne Argumente auf. Das Ergebnis ist ein lauffähiges Programm mit Namen `fehler`. Der Aufruf von `fehler` führt bei sinnvollen Eingaben zu einer Ausgabe, die richtig sein könnte. Wir haben aber noch einen Denkfehler darin. Statt der Summe wird der Integer-Quotient berechnet. Wir berichtigen auch das und testen das Programm mit einigen Eingaben. Da unser Quelltext richtig zu sein scheint, verschönern wir seine vorläufig endgültige Fassung mit dem Beautifier `cb(1)`:

```
cb fehler.c > fehler.b
rm fehler.c
mv fehler.b fehler.c
```

Schließlich löschen wir das nicht mehr benötigte Corefile und untersuchen das Programm noch mit einigen Werkzeugen:

```
time fehler
cflow fehler.c
cxref fehler.c
strings fehler
nm fehler
size fehler
ls -l fehler
strip fehler
ls -l fehler
```

`strings(1)` ist ein ziemlich dummes Werkzeug, das aus einer ausführbaren Datei alles heraussucht, was nach String aussieht. Das Werkzeug `nm(1)` gibt eine Liste aller Symbole aus, die lang werden kann. `strip(1)` wirft aus einer ausführbaren Datei die nur für den Debugger, nicht aber für die Ausführung wichtigen Informationen heraus und verkürzt dadurch die Datei. Abmelden mit `exit`.

Schreiben Sie in einer Programmiersprache Ihrer Wahl (ich empfehle C) ein Programm, das

- eine Datei mittels `creat(2)` erzeugt,
- dessen Zugriffsrechte mittels `chmod(2)` und seine Zeitstempel mittels `utime(2)` setzt,
- die verwendeten Werte mittels `fprintf(3)` als Text in die Datei schreibt. `fprintf(3)` finden Sie unter `printf(3)`.

Schreiben Sie ein Programm ähnlich `who(1)`. Sie brauchen dazu `getut(3)` und `utmp(4)`.

1.2.13 Fragen Programmer's Workbench

- Wozu braucht man einen Compiler? Einen Linker?
- Was ist `lint`?
- Was macht `make`? Wie sieht ein einfaches Makefile aus?
- Wozu braucht man Debugger?
- Was ist eine Funktionsbibliothek? Vorteil?
- Wozu braucht man eine Versionsverwaltung? Wie benutzt man RCS?
- Was sind Systemaufrufe? Wer braucht sie?
- Unterschied zu Standardfunktionen?
- Welche Aufgaben erledigen die Systemaufrufe hauptsächlich?

1.3 Bausteine eines Quelltextes

1.3.1 Übersicht

Alle Zeichen oder Zeichengruppen eines Programmes im Quellcode sind entweder

- Kommentar (comment),
- Namen (identifier),
- Schlüsselwörter (Wortsymbole) (keyword),
- Operatoren (operator),
- Konstanten (Literale) (constant, literal),
- Trennzeichen (separator) oder
- bedeutungslos.

Ein Quelltext (Programmquelle, Source) ist ein **Textfile**, bestehend aus Zeilen, diese wiederum aus druck- und sichtbaren Zeichen (fast immer des US-ASCII-Zeichensatzes) einschließlich des Zwischenraums (space). Die Zeichen sind zu Wörtern gruppiert, die durch Trennzeichen voneinander abgetrennt sind, meist durch mindestens einen Zwischenraum. Zeilen werden durch das Zeilenende markiert, meist durch die ASCII-Zeichen CR und/oder LF. Sinneinheiten wie Wörter, Zahlen oder Operatoren werden als **Token** bezeichnet, sie dürfen nicht durch Trennzeichen oder Zeilenwechsel unterbrochen werden.

Kommentar ist nur für den menschlichen Leser bestimmt und gelangt in C/C++ gar nicht bis zum Übersetzungsvorgang, sondern wird schon vom Präprozessor entfernt und kann bis auf seine Begrenzungen frei gestaltet werden. Ebenso entfernt der Präprozessor das Zeichenpaar Backslash-Zeilenwechsel und verbindet somit zwei Zeilen. Auf diese Weise lassen sich lange Anweisungen auf mehrere Zeilen verteilen. **Schlüsselwörter** und **Operatoren** sind festgelegte Zeichen oder Zeichengruppen, an die jeder gebunden ist. **Namen** werden nach gewissen Regeln vom Programmierer gebildet, ebenso **Konstanten**. **Trennzeichen** trennen die genannten Bausteine oder ganze Anweisungen voneinander und sind festgelegt, meist Leerzeichen (space), Semikolons und Linefeeds. Bedeutungslose Zeichen sind überzählige Leerzeichen, Tabs oder Linefeeds.

Der C-Standard – gegenwärtig ISO/IEC 9899:1999 – legt nicht alle Einzelheiten fest. Zu manchen Fragen spricht er nur Empfehlungen aus oder überlässt sogar die Wahl dem Compiler und/oder dem Betriebssystem. Gerät man an solche Probleme, braucht man eine genaue Beschreibung der beiden, sollte experimentell prüfen, was geschieht, und sein Programm ausführlich kommentieren.

1.3.2 Syntax-Diagramme

Die Syntax der einzelnen Bausteine – das heißt ihr regelgerechter Gebrauch – kann mittels Text beschrieben werden. Das ist oft umständlich und teilweise schwer zu verstehen. Deshalb nimmt man Beispiele zu Hilfe, die aber selten die Syntax vollständig erfassen. So haben sich **Syntax-Diagramme** eingebürgert, die nach etwas Übung leicht zu lesen sind. In Abbildung 1.3 auf Seite 47 ist die Syntax zweier C-Bausteine dargestellt, nämlich die `if-else`-Anweisung und den Block. Die `if-else`-Anweisung besteht aus:

- dem Schlüsselwort `if`,
- einer öffnenden runden Klammer,
- einem booleschen Ausdruck (true – false),
- einer schließenden runden Klammer,
- einer Anweisung – auch die leere Anweisung – oder einem Block,
- dann ist entweder Ende der `if`-Anweisung oder es folgt
- das Schlüsselwort `else`, gefolgt von

if-else-Anweisung

if (bool. Ausdr.) Anw. Block
else Anw. Block

Block

{ Anweisung }

Abb. 1.3: Syntax-Diagramm der if-else-Anweisung und des Blockes

- einer Anweisung oder einem Block.

Ist der boolesche Ausdruck `true`, dann wird die erste Anweisung ausgeführt, andernfalls die zweite. Fehlt der `else`-Zweig, so liegt eine *bedingte Anweisung* vor, die nur dann ausgeführt wird, wenn der boolesche Ausdruck `true` ist. Danach geht es im Programm weiter.

Ein Block seinerseits besteht aus:

- einer öffnenden geschweiften Klammer,
- dann entweder nichts (leerer Block) oder
- einer Anweisung,
- gegebenenfalls weiteren Anweisungen,
- und einer schließenden geschweiften Klammer.

Da ein Block syntaktisch gleichwertig einer Anweisung ist, lassen sich Blöcke schachteln. Ein Block kann überall stehen, wo eine Anweisung erwartet wird.

Ein weiterer Weg zur Beschreibung der Syntax einer Programmiersprache ist die Backus-Naur-Form, die von JOHN BACKUS, einem der Väter von FORTRAN, und PETER NAUR, einem der Väter von ALGOL, als Metasprache zu ALGOL 60 entwickelt worden ist. Weiteres siehe bei D. GRIES.

1.3.3 Kommentar

Alle Programmiersprachen ermöglichen, Text in ein Programm einzufügen, der vom Compiler überlesen wird und nur für den menschlichen Leser bestimmt ist. Dieser **Kommentar** muss mit einem besonderen Zeichen eingeleitet und gegebenenfalls beendet werden.

In C/C++ leitet die Zeichengruppe `/*` den Kommentar ein. Er kann sich über mehrere Zeilen erstrecken, darf aber nicht geschachtelt werden. Zu einer ungewollten Schachtelung kommt es, wenn man kommentierte Programmteile durch Einrahmen mit Kommentarzeichen vorübergehend unwirksam macht. Die Fehlermeldung des Compilers sagt irgendetwas mit Pointern und führt irre. Die Zeichengruppe `*/` kennzeichnet das Ende. Ein Zeilenende beendet diesen Kommentar nicht. Ansonsten kann Kommentar überall stehen, nicht nur auf einer eigenen Zeile. Ein Beispiel:

```
/*
Die ersten Zeilen enthalten Programmnamen, Zweck, Autor,
Datum, Compiler, Literatur und aehnliches.
*/

#include <stdio.h>

int main()
{
/* Dies ist eine eigene Kommentarzeile */
puts("Erste Zeile");
puts("Zweite Zeile");    /* Kommentar */
/* Kommentar */     puts("Dritte Zeile");

/*
puts("Vierte Zeile");    /* Kommentar geschachtelt!!! */
*/

puts("Ende");
return 0;
}
```

Quelle 1.21 : C-Programm mit Kommentaren

Auf unserem System haben wir folgende Regel eingeführt: da Fehlermeldungen des Systems in Englisch ausgegeben werden, schreiben wir die Meldungen unserer Programme in Deutsch (man sollte sie ohnedies mittels `#define`-Anweisungen irgendwo zusammenfassen, so dass sie leicht ausgetauscht werden können). Damit sieht man sofort, woher eine Meldung stammt. Kommentar schreiben wir wieder in Englisch, da die Programmbeispiele auch per Mail oder News in die unendlichen Weiten des Internet geschickt werden, wo Englisch nun einmal die lingua franca ist. An Kommentar[21] soll man nicht sparen, denn er kostet wenig Aufwand und kann viel helfen, während die Dokumentation zum Programm nur zu oft ad calendas Graecas (Sankt-Nimmerleins-Tag) verschoben wird.

In C++ kommt eine weitere Art von Kommentar hinzu. Er beginnt mit zwei Schrägstrichen und endet mit dem Zeilenwechsel, weshalb er **Zeilenkommentar** genannt wird. Dieser Kommentar darf innerhalb des oben genannten Kommentars vorkommen. Die umgekehrte Folge ist auch zulässig, aber selten anzutreffen.

1.3.4 Namen

Namen (identifier) bezeichnen Funktionen, Konstanten, Variable, Makros oder Sprungmarken (Labels). Sie müssen mit einem Buchstaben oder einem Unterstrich (underscore) beginnen. Benutzereigene Namen sollten immer mit einem Buchstaben anfangen, der Unterstrich als erstes Zeichen wird vom Compiler oder vom System verwendet. Die weiteren Zeichen des Namens können Buchstaben, Ziffern oder der Unterstrich sein. Groß- und Kleinbuchstaben werden unterschieden. Die maximal zulässige Länge von Namen kann durch den Compiler, den Linker oder das Bestriebssystem gegeben sein und lässt sich daher nicht allgemein angeben. Ab 255 Zeichen wird es kritisch. Signifikant sind mindestens die ersten sieben Zeichen, nach ANSI die ersten einunddreißig. Verwendet man Funktionen fremder Herkunft, sollte man mit nur sechs signifikanten Zeichen rechnen sowie damit, dass Groß- und Kleinbuchstaben *nicht* unterschieden werden.

1.3.5 Schlüsselwörter

In C/C++ wie in jeder anderen Programmiersprache haben bestimmte Wörter eine besondere Bedeutung, beispielsweise `main`, `while` und `if`. Diese **Wortsymbole** oder **Schlüsselwör-**

[21]Real programmers don't comment their code.

ter[22] dürfen auf keinen Fall als Namen verwendet werden, die Namen der Standardfunktionen wie `printf()` oder `fopen()` sollen nicht umfunktioniert werden. C zeichnet sich durch eine geringe Anzahl von Schlüsselwörtern aus, etwa dreißig, siehe Anhang D.1 *C-Lexikon, Schlüsselwörter* auf Seite 198. Mit C++ kommen nochmal dreißig dazu. Unterboten wird C/C++ darin nur von SMALLTALK mit fünf Schlüsselwörtern.

1.3.6 Operanden

Wir schränken hier den Begriff *Daten* etwas ein und verstehen darunter nur die passiven Objekte, mit denen ein Programm etwas tut, also Text, Zahlen, Grafiken usw. Diese Objekte und ihre Untereinheiten nennen wir **Operanden** (operand). Mit ihnen werden Operationen durchgeführt. Das Wort *Objekt*[23] vermeide ich hier, um keine Assoziationen an objektorientiertes Programmieren zu wecken. Ein Operand

- hat einen **Namen** (identifier),
- gehört einem **Typ** (type) an,
- hat einen konstanten oder variablen **Wert** (value),
- belegt zur Laufzeit **Speicherplatz** im Computer,
- hat einen **Geltungsbereich** (scope) und
- eine **Lebensdauer** (lifetime).

Auf den Operanden wird über den Namen oder die Speicheradresse zugegriffen. Die Speicheradresse eines Operanden kann in einem weiteren Operanden abgelegt werden, der Zeiger, Referenz, Adressvariable oder **Pointer** genannt wird. Ich bevorzuge das englische Wort *Pointer*, weil das deutsche Wort *Zeiger* drei Bedeutungen hat: Pointer, Index, Cursor. Außerdem bezeichne ich Pointer nicht als Variable, obwohl ihr Wert veränderlich ist. Wer das englische Wort nicht mag, sollte von Adressvariablen reden. Pointer auf Pointer sind Pointer 2. Ordnung usw. Der Geltungsbereich eines Operanden ist ein Block, eine Funktion, eine Datei oder das ganze Programm. Gleiches gilt für die Lebensdauer. In der **Deklaration** eines Operanden werden sein Name und seine Eigenschaften vereinbart. In der **Definition** erhält er einen Wert und benötigt spätestens dann einen Platz im Arbeitsspeicher. Deklaration und Definition können in *einer* Anweisung zusammengezogen sein. Die erstmalige Zuweisung eines Wertes an eine Variable heißt **Initialisierung**. Deklaration und Definition werden auch unter dem Begriff **Vereinbarung** zusammengefasst.

Auf die Auswahl und Strukturierung der Operanden soll man Sorgfalt verwenden. Eine zweckmäßige **Datenstruktur** erleichtert das Programmieren und führt zu besseren Programmen. Eine nachträgliche Änderung der Datenstruktur erfordert meist einen großen Aufwand, weil viele Programme oder Programmteile davon betroffen sind. Die Namen der Operanden sollen ihre Bedeutung erklären, erforderlichenfalls ist ihre Bedeutung im Kommentar oder in einer Aufzählung festzuhalten.

1.3.6.1 Konstanten und Variable

Operanden können während des Ablaufs eines Programmes konstant bleiben (wie die Zahl π) oder sich ändern (wie die Anzahl der Iterationen zur Lösung einer Gleichung oder das

[22] Es gibt die Bezeichnungen Wortsymbol, Schlüsselwort und reserviertes Wort. Gemeint ist in jedem Fall, dass das Wort – eine bestimmte Zeichenfolge – nicht uneingeschränkt als Namen verwendet werden darf. In C dürfen diese Wörter – außer im Kommentar – keinesfalls für einen anderen als ihren besonderen Zweck verwendet werden. In FORTRAN dürfen diese Wörter in Zusammenhängen, die eine Deutung als Schlüsselwort ausschließen, auch als Namen verwendet werden. Man darf also eine Variable `if` nennen, und in der Zuweisung `if = 3` wird die Zeichenfolge `if` als Variable und nicht als Wortsymbol im Sinne von *falls* verstanden.

[23] KERNIGHAN + RITCHIE gebrauchen *Objekt* im Sinne eines Speicherbereiches, auf den mittels eines Namens zugegriffen wird.

Ergebnis einer Berechnung oder Textsuche). Es kommt auch vor, dass ein Operand für einen Programmaufruf konstant ist, beim nächsten Aufruf aber einen anderen Wert hat (wie der Mehrwertsteuersatz).

Man tut gut, sämtliche Operanden eines Programmes an wenigen Stellen zusammenzufassen und zu deklarieren. In den Funktionen oder Prozeduren sollen keine geheimnisvollen Zahlen (magic numbers) auftauchen, sondern nur Namen. Konstanten, die im Programm über ihren Namen aufgerufen werden, heißen **symbolische Konstanten**.

Für den Computer sind Konstanten Bestandteil des Programmcodes, das unter UNIX in das Codesegment des zugehörigen Prozesses kopiert und vor schreibenden Zugriffen geschützt wird. Diese Konstanten werden auch **Literale** genannt. Variable hingegen belegen Speicherplätze im User Data Segment, deren Adressen das Programm kennt und auf die es lesend und schreibend zugreift.

In ANSI-C sind die **Typ-Attribute** (type qualifier) `const` und `volatile` eingeführt worden, die eine bestimmte Behandlung der zugehörigen Operanden erzwingen. Sie werden selten gebraucht.

1.3.6.2 Typen – Grundbegriffe

Jeder Operand gehört einem **Typ** an, der über

- den Wertebereich (siehe `/usr/include/limits.h`),
- die zulässigen Operationen,
- den Speicherbedarf

entscheidet. Die Typen werden in drei Gruppen eingeteilt:

- einfache, skalare oder elementare Typen
- zusammengesetzte oder strukturierte Typen
- Pointer (Adressvariable)

In C gibt es nur konstante Typen, das heißt, ein Operand, der einmal als ganzzahlig deklariert worden ist, bleibt dies während des ganzen Programmes. Einige Programmiersprachen erlauben auch variable Typen, die erst zur Laufzeit bestimmt werden oder sich während dieser ändern. Typfreie Sprachen kennen nur das Byte oder das Maschinenwort als Datentyp. Die Typisierung[24] erleichtert die Arbeit und erhöht die Sicherheit sowie die Rechengeschwindigkeit. Stellen Sie sich vor, Sie müssten bei Gleitkommazahlen Exponent und Mantisse jedesmal selbst aus den Bytes herausdröseln. Oder der Computer müsste jedesmal aus dem Zusammenhang einer Operation ermitteln, um was für einen Typ von Operanden es sich handelt. Es gibt aber auch Aufgaben, bei denen der Verzicht auf eine Typisierung Vorteile bringt. Oft ist das bei Aufgaben der Fall, die mittels Skriptsprachen bearbeitet werden.

Die Typdeklarationen in C/C++ können ziemlich schwierig zu verstehen sein, vor allem bei mangelnder Übung. Im Netz findet sich ein Programm `cdecl`, das Typdeklarationen in einfaches Englisch übersetzt. Füttert man dem Programm folgende Deklaration:

```
char (*(*x[3]) ()) [5]
```

so erhält man zur Antwort:

```
declare x as array 3 of pointer to function returning
pointer to array 5 of char
```

So schnell wie `cdecl` hätte ich die Antwort nicht gefunden.

1.3.6.3 Einfache Typen

In jeder Programmiersprache gibt es Grundtypen, aus denen alle höheren Typen zusammengesetzt werden. In C/C++ sind dies ganze Zahlen, Gleitkommazahlen und Zeichen.

[24]Real programmers don't worry about types.

Ganze Zahlen In C/C++ gibt es ganze Zahlen mit oder ohne Vorzeichen sowie in halber, einfacher oder doppelter Länge:

- `int` ganze Zahl mit Vorzeichen
- `unsigned int` ganze Zahl ohne Vorzeichen
- `short` kurze ganze Zahl mit Vorzeichen
- `unsigned short` kurze ganze Zahl ohne Vorzeichen
- `long` ganze Zahl doppelter Länge mit Vorzeichen
- `unsigned long` ganze Zahl doppelter Länge ohne Vorzeichen

Die Deklaration von Variablen als ganzzahlig sieht so aus:

```
int x, y, z;
unsigned long anzahl;
```

Die Länge der ganzen Zahlen in Bytes ist nicht festgelegt und beim Portieren zu beachten. Häufig sind `int` und `long` gleich und belegen ein **Maschinenwort**, auf unserer Anlage also 4 Bytes gleich 32 Bits (32-Bit-Architektur). Festgelegt ist nur die Reihenfolge:

```
char <= short <= int <= long
```

Alle Annahmen, die darüber hinausgehen, sind Vermutungen, die auf einer Maschine zutreffen, auf einer anderen nicht. Auch die Annahme, dass die Typen `int` und `Pointer` immer gleich viele Bytes belegen, ist schierer Aberglaube.

Mit dem Aufkommen von 64-Bit-Maschinen ist die Diskussion der Länge von Datentypen neu entfacht. Beim Übergang von 16 auf 32 Bit gegen Ende der siebziger Jahre waren der Kreis der UNIX- und C-Programmierer und damit das ganze Problem wesentlich kleiner als heute. Wenn 128-Bit-Maschinen die Regel werden, wird sich die Diskussion nochmals wiederholen, allerdings aufbauend auf den Erfahrungen des gegenwärtigen Wechsels. Das Ziel sind Programme, die unabhängig von der Datengröße auf allen Architekturen laufen, und Daten, die zwischen verschiedenen Architekturen ausgetauscht werden können.

Datentyp	LP32	ILP32	ILP64	LLP64	LP64
	2/4/4	4/4/4	8/8/8	4/4/8	4/8/8
char	8	8	8	8	8
short	16	16	16	16	16
int	16	32	64	32	32
long	32	32	64	32	64
Pointer	32	32	64	64	64

Tab. 1.1: Länge von Datentypen auf verschiedenen Architekturen

Je nach Länge der Datentypen `int` (I), `long` (L) und `Pointer` (P) unterscheidet man heute die in Tabelle 1.1 auf Seite 51 aufgeführten Architekturen. Es würde zu weit führen, hier die Vor- und Nachteile jeder Architektur gegeneinander abzuwägen. Wichtig ist, die Architektur der eigenen Maschine zu kennen (in unserem Fall ILP32) und die Programme möglichst portabel zu gestalten. Hierzu Empfehlungen im Abschnitt 1.13 *Portieren von Programmen* auf Seite 173.

Für ganze Zahlen sind die Addition, die Subtraktion, die Multiplikation, die Modulo-Operation (Divisionsrest) und die Division unter Vernachlässigung des Divisionsrestes definiert, ferner Vergleiche mittels größer – gleich – kleiner.

Gleitkommazahlen Gleitkommazahlen – auch als Reals oder Floating Point Numbers bezeichnet – werden durch eine **Mantisse** und einen **Exponenten** dargestellt. Der Exponent versteht sich nach außen zur Basis 10, intern wird die Basis 2 verwendet. Die Mantisse ist auf eine Stelle ungleich 0 vor dem Dezimalkomma oder -punkt normiert. Es gibt:

- `float` Gleitkommazahl einfacher Genauigkeit
- `double` Gleitkommazahl doppelter Genauigkeit
- `long double` Gleitkommazahl noch höherer Genauigkeit (extended precision)

Die Deklaration von Gleitkomma-Variablen sieht so aus:

```
float x, y, z;
double geschwindigkeit;
```

Gleitkommazahlen haben immer ein Vorzeichen. Man beachte, daß die Typen sich nicht nur in ihrem Wertebereich, sondern auch in ihrer Genauigkeit (Anzahl der signifikanten Stellen) unterscheiden, anders als bei Ganzzahlen. Der Typ `long double` ist selten.

Für Gleitkommazahlen sind die Addition, die Subtraktion, die Multiplikation, die Division sowie Vergleiche zulässig. Die Abfrage auf Gleichheit ist jedoch heikel, da aufgrund von Rundungsfehlern zwei Gleitkommazahlen selten gleich sind. Wenn möglich, mache man um Gleitkommazahlen einen großen Bogen. Die Operationen dauern länger als die entsprechenden für Ganzzahlen, und die Auswirkungen von Rundungsfehlern sind schwierig abzuschätzen. Zur internen Darstellung von Gleitkommazahlen siehe Abschnitt **??** *Arithmetikprozessoren* auf Seite **??**.

Alphanumerischer Typ Eine Größe, deren Wertevorrat die Zeichen der ASCII-Tabelle oder einer anderen Tabelle sind, ist vom Typ **alphanumerisch** oder **character**, bezeichnet mit `char`. In C werden sie durch eine Integerzahl zwischen 0 und 127 (7-bit-Zeichensätze) beziehungsweise 255 (8-bit-Zeichensätze) dargestellt. Der Speicherbedarf beträgt ein Byte. Mittlerweile gibt es auch internationale Zeichensätze, deren Zeichen je zwei Bytes belegen. Die Deklaration von alphanumerischen Variablen sieht so aus:

```
char a, b, c;
```

Mit wachsender Verbreitung von 16-bit-Zeichensätzen (Unicode, Intercode) ist zu erwarten, dass die Länge des `char`-Typs angepasst wird.

Die Verwandtschaft zwischen Ganzzahlen und Zeichen in C verwirrt anfangs. Man mache sich die Gemeinsamkeiten an einem kleinen Programm klar:

```
/* Programm zum Demonstrieren von character und integer */

#include <stdio.h>

int main()
{
int  i, j, k; char a, b;

i = 65; j = 233; k = 333; a = 'B'; b = '!';

printf("Ganzzahlen: %d  %d  %d  %d\n", i, j, k, a);
printf("Zeichen   : %c  %c  %c  %c\n", i, j, k, a);

puts("Nun rechnen wir mit Zeichen (B = 66, ! = 33):");
printf("%c + %c = %d\n", a, b, a + b);
printf("%c - %c = %d\n", b, a, b - a);
printf("%c - %c = %c\n", b, a, b - a);

return 0;
}
```

Quelle 1.22 : C-Programm mit den Typen character und integer

Die Ausgabe des Programms lautet:

```
Ganzzahlen: 65  233  333  66
Zeichen   : A  é  M  B
Nun rechnen wir mit Zeichen (B = 66, ! = 33):
B + ! = 99
! - B = -33
! - B = SS
```

In der ersten Zeile werden alle Werte entsprechend dem Formatstring der Funktion `printf(3)` als dezimale Ganzzahlen ausgegeben, wobei der Buchstabe `B` durch seine ASCII-Nummer 66 vertreten ist. In der zweiten Zeile werden alle Werte als 7-bit-ASCII-Zeichen verstanden, wobei die Zahlen, die mehr als 7 Bit (> 127) beanspruchen, nach 7 Bit links abgeschnitten werden. Die Zahl 233 führt so zur Ausgabe des Zeichens Nr. 233 - 128 = 105. Die Zahl - 33 wird als Zeichen Nr. 128 - 33 = 95, dem Unterstrich, ausgegeben. Wie man an der Rechnung erkennt, werden Zeichen vom Prozessor wie ganze Zahlen behandelt und erst bei der Ausgabe einer Zahl oder einem ASCII-Zeichen zugeordnet. Es ist zu erwarten, daß auf Systemen mit 8-bit- oder 16-bit-Zeichensätzen die Grenze höher liegt, aber die Arbeitsweise bleibt. Manchmal will man ein Byte wahlweise als Ganzzahl oder als Zeichen auffassen, aber das gehört zu den berüchtigten Tricks in C. Meint man den Buchstaben a, sollte man auch `'a'` schreiben, denn der Gebrauch der Nummer 97 anstelle des Zeichens setzt voraus, dass das System, auf dem das Programm ausgeführt wird, den ASCII-Zeichensatz verwendet, womit die Portabilität des Programms eingeschränkt wird.

Das Ausgabegerät empfängt nur die Nummer des auszugebenden Zeichens gemäß seiner Zeichensatz-Tabelle (ASCII, ROMAN8), die Umwandlung des Wertes entsprechend seinem Typ ist Aufgabe der Funktion `printf(3)`.

Boolescher Typ Eine Größe vom Typ `boolean` oder `logical` kann nur die Werte `true` (wahr, richtig) oder `false` (falsch) annehmen. In C werden statt des booleschen Typs die Integerwerte 0 (= false) und nicht-0 (= true) verwendet. Verwirrend ist, dass viele Funktionen bei Erfolg den Wert 0 und bei Fehlern Werte ungleich 0 zurückgeben.

Leerer Typ Der leere Datentyp **void** wird zum Deklarieren von Funktionen verwendet, die kein Ergebnis zurückliefern, sowie zum Erzeugen generischer (allgemeiner) Pointer, die auf Variablen eines vorläufig beliebigen Typs zeigen. Der Typ hat keine Größe. Der Bytebedarf eines Pointers dagegen liegt fest, auch wenn der zugehörige Variablentyp noch offen ist. Zur Pointer-Arithmetik muss jedoch der Typ (das heißt der Bytebedarf der zugehörigen Variablen) bekannt sein. Variablen vom Typ `void` lassen sich nicht verarbeiten, weil sie nicht existieren. Man bezeichnet den Typ `void` als unvollständig, da er nicht alle Fähigkeiten eines vollständigen Typs wie `int` aufweist. Man braucht diesen schrägen Typ aus ähnlichen Gründen wie die leere Menge in der Mathematik.

Vor der Erfindung des Typs `void` wurde für generische Pointer der Typ `char` genommen, der ein Byte umfasst, woraus sich alle anderen Typen zusammensetzen lassen. Man hätte den Typ auch `byte` nennen können.

1.3.6.4 Zusammengesetzte Typen (Arrays, Strukturen)

Arrays Die meisten Programmiersprachen kennen **Arrays**, auch als Vektoren oder unglücklicherweise als **Felder**[25] bezeichnet; das sind geordnete Mengen von Größen desselben Typs. Jedem Element ist ein fortlaufender Index (Hausnummer) zugeordnet, der in C stets mit 0 beginnt. In einem Array von zwölf Elementen läuft also der Index von 0 bis 11, aufpassen.

Elemente eines Arrays dürfen Konstanten oder Variable aller einfachen Typen, andere Arrays, Strukturen, Unions oder Pointer sein, jedoch keine Funktionen. Dateien sind formal

[25] Felder in Datensätzen sind etwas völlig anderes.

Strukturen, ein Array von Dateien ist also erlaubt. Die Deklaration von Arrays sieht folgendermaßen aus:

```
int zahlen[100], nr[12];
int matrix[4][3];
double realnumbers[1000];
char names[33];
char zeichen[] = "abcd";
```

Der Compiler muss die Größe eines Arrays (Anzahl und Typ der Elemente) wissen. Sie muss bereits im Programm stehen und kann nicht erst zur Laufzeit errechnet werden. Man kann jedoch die Größe eines Arrays zur Laufzeit mittels der Standardfunktion `malloc(3)` ändern, siehe Abschnitt 1.11.9 *Dynamische Speicherverwaltung* auf Seite 158.

Es gibt mehrdimensionale Arrays (Matrizen usw.) mit entsprechend vielen Indexfolgen. Die Elemente werden im Speicher hintereinander in der Weise abgelegt, dass sich der letzte Index am schnellsten ändert. Der Compiler linearisiert das Array, wie man sagt. Eine Matrix wird zeilenweise gespeichert. Vorsicht beim Übertragen von oder nach FORTRAN: dort läuft die Indizierung anders als in C/C++, eine Matrix wird spaltenweise gespeichert. PASCAL verhält sich wie C/C++.

Der **Name** eines Arrays ist eine Adresskonstante und kann daher nicht auf der linken Seite einer Zuweisung vorkommen. Weiteres dazu im Abschnitt 1.3.6.7 *Pointer* auf Seite 56.

Strings (Zeichenketten) In C/C++ sind **Strings** oder **Zeichenketten** (chaine de caractères) Arrays of characters, abgeschlossen durch das ASCII-Zeichen Nr. 0 (nicht zu verwechseln mit der Ziffer 0 entsprechend ASCII-Nr. 48). Strings dürfen nicht beliebig lang werden. Wenn nicht Arbeitsspeicher, Editor oder andere Faktoren vorher zuschlagen, muss man ab 32 kByte auf Probleme gefasst sein, wohlgmerkt beim einzelnen String, nicht bei einem aus vielen Strings bestehenden Text. Man stößt selten an diese Grenze, deshalb wird sie in vielen Büchern nicht erwähnt.

Wir bevorzugen das Wort *String* um hervorzuheben, dass es sich hierbei um Zeichenfolgen in einem bestimmten, sprachenspezifischen Format handelt. Zum Speichern des Strings `Alex` ist ein Array of characters mit wenigstens fünf Elementen zu deklarieren:

```
char myname[5];
```

In anderen Sprachen werden Strings anders dargestellt. Ein String lässt sich am Stück verarbeiten oder durch Zugriff auf seine Elemente. Man kann fertige String-Funktionen verwenden oder eigene Funktionen schreiben, muss sich dann aber auch selbst um die ASCII-Null kümmern.

Will man bei der Eingabe von Werten mittels der Tastatur jeden beliebigen Unsinn zulassen, dann muss man die Eingaben als lange (einige Zeilen) Strings übernehmen, die Strings prüfen und dann – sofern sie vernünftig sind – in den gewünschten Typ umwandeln. Ein gutes Programm vertraut Eingaben niemals blindlings, sondern prüft sie vor der weiteren Verarbeitung gründlich. Ein Programmbeispiel dazu findet sich im Abschnitt 1.11.7.2 *Pointer auf Typ void: xread.c* auf Seite 148.

Merke: Es gibt Arrays of characters, die keine Strings sind, nämlich solche, die nicht mit dem ASCII-Zeichen Nr. 0 abgeschlossen sind. Sie müssen als Array angesprochen werden wie ein Array von Zahlen.

Merke zweitens: Ein einzelnes Zeichen kann als Zeichen (character, `'a'`) oder als String (array of characters, `"a"`) dargestellt werden. Für ein Programm sind das verschiedene Dinge.

Strukturen Eine **Struktur**, auch als **Verbund** und in PASCAL als **Record** bezeichnet, vereint Komponenten ungleichen Typs im Gegensatz zum Array. Eine Ordnung der Komponenten liegt nicht vor, ebenfalls anders als beim Array. Strukturen dürfen geschachtelt werden, aber nicht sich selbst enthalten (keine Rekursion). Möglich ist jedoch, dass eine Struktur einen Pointer auf sich selbst enthält – ein Pointer ist ja nicht die Struktur selbst – womit Verkettungen hergestellt werden. Das Schlüsselwort lautet `struct`.

Ein typisches Beispiel für eine Struktur ist eine Personal- oder Mitgliederliste, bestehend aus alphanumerischen und numerischen Komponenten. Mit den numerischen wird gerechnet, auf die alphanumerischen werden Stringfunktionen angewendet. Telefonnummern oder Postleitzahlen sind alphanumerische Größen, da Rechenoperationen mit ihnen sinnlos sind. Wir erzeugen einen Strukturtyp ohne eigenen Namen und deklarieren zugleich eine Variable namens `mitglied`:

```
struct {
char nachname[32];
char vorname[32];
int  beitrag;
       } Mitglied;
```

Man kann auch zuerst nur die Struktur definieren und in einem zweiten Schritt Variablen vom Typ dieser Struktur:

```
struct mg {
   char nachname[32];
   char vorname[32];
   int  beitrag;
  };

struct mg Mitglied;
```

Jede Datei ist eine Struktur namens `FILE`, die in der include-Datei `stdio.h` deklariert ist:

```
typedef struct {
        int     _cnt;
        unsigned char    *_ptr;
        unsigned char    *_base;
        short   _flag;
        char    _file;
} FILE;
```

Mit dieser `typedef`-Deklaration wird ein Strukturname `FILE` vereinbart, der in weiteren Deklarationen als Typ auftritt. `FILE` ist keine Variable, sondern ein Synonym für obige Struktur. Anschließend lassen sich Variable vom Typ `FILE` oder auch Dateipointer deklarieren:

```
FILE myfile, yourfile;
FILE *fp;
```

Dies ist ein dritter Weg, den wir im Abschnitt *Weitere Namen für Typen* auf Seite 61 kennenlernen.

In C/C++ sind alle Dateien ungegliederte Folgen von Bytes (Bytestreams), so dass es keinen Unterschied zwischen Textfiles und sonstigen Dateien gibt. Die Gliederung erzeugt das lesende oder schreibende Programm. Anders als in PASCAL ist daher der Typ `FILE` nicht ein `FILE of irgendetwas`.

Eine besondere Struktur ist das **Bitfeld**. Die Strukturkomponenten sind einzelne Bits oder Gruppen von Bits, die über ihren Komponentennamen angesprochen werden. Eine Bitfeld-Struktur darf keine weiteren Komponenten enthalten und soll möglichst vom Basistyp `unsigned` sein. Ein einzelnes Bitfeld darf maximal die Länge eines Maschinenwortes haben, es kann also nicht über eine Wortgrenze hinausragen. Bitfelder sind keine Arrays, es gibt keinen Index. Ebensowenig lassen sich Bitfelder referenzieren (&-Operator). Bitfelder werden verwendet, um mehrere Ja-nein-Angaben in einem Wort unterzubringen.

Der Name einer Strukturvariablen ist ein gewöhnlicher Variablenname, *kein* Pointer.

1.3.6.5 Union

Eine Variable des Typs `union` kann Werte unterschiedlichen Typs aufnehmen, zu einem Zeitpunkt jedoch immer nur einen. Es liegt in der Hand des Programms, über den augenblicklichen Typ Buch zu führen. In FORTRAN dient die equivalence-Anweisung demselben Zweck, in PASCAL der variante Record. Eine Union belegt so viele Bytes wie der längste in ihr untergebrachte Datentyp. Die Deklaration einer Variablen als Union sieht aus wie bei einer Struktur:

```
union unioneins {
         int i;
         double x;
         char c;
         } ux;
```

Damit wird ein Unionstyp mit dem Namen `unioneins` deklariert und zugleich eine Variable `ux` dieses Typs. Auf die jeweilige Variable wird zugegriffen wie auf die Komponenten einer Struktur:

```
printf("%d\n", ux.i);
printf("%f\n", ux.x);
printf("%c\n", ux.c);
```

Man darf nur jeweils die Variable herausholen, die als letzte hineingesteckt worden ist, andernfalls gibt es Überraschungen. Die Union habe ich noch nie gebraucht, sie soll in der Systemprogrammierung vorkommen und trägt sicher nicht zur Klarheit eines Programmes bei.

1.3.6.6 Aufzählungstypen

Durch Aufzählen lassen sich benutzereigene Typen schaffen. Denkbar ist:

```
enum wochentag {montag, dienstag, mittwoch, donnerstag,
                freitag, samstag, sonntag} tag;
```

Die Variable `tag` ist vom Typ `wochentag` und kann die oben aufgezählten Werte annehmen. Die Reihenfolge der Werte ist maßgebend für Vergleiche: `montag` ist kleiner als `dienstag`. Auch Farben bieten sich für einen Aufzählungstyp an. In der Maschine werden Aufzählungstypen durch Ganzzahlen dargestellt, insofern handelt es sich nicht um einen neuen Typ. Aufzählungstypen verbessern die Lesbarkeit der Programme.

1.3.6.7 Pointer (Zeiger)

Pointer sind vom Geheimnis umwittert. Wer mit Pointern umgehen kann (oder wenigstens so tut), verfügt über magische Kräfte und steht mit finsteren Mächten im Bunde. Vermutlich frisst er auch kleine Kinder oder unschuldige Windows-Benutzer. Dabei ist das Arbeiten mit Pointern einfach, man muss nur langsam und klar denken und saubere Begriffe verwenden.

Pointer erweitern die Möglichkeiten des Programmierers. Ein Beispiel für den sinnvollen Einsatz von Pointern: das Sortieren von Datensätzen. Die Datensätze enthalten in Form einer Struktur Namen, Anschrift und weitere Angaben zu Personen. Die Datensätze sollen nach dem Namen sortiert werden, um einen schnellen Zugriff zu ermöglichen. Würden wir die Datensätze nach Hinzufügen oder Löschen einzelner Sätze jedesmal neu sortieren, so wäre das mit viel Kopierarbeit verbunden. Stattdessen legt man ein Hilfsfile, einen Index an, der nur die Sortierschlüssel (Namen) und die Speicheradressen (Pointer) der zugehörigen Datensätze enthält. Dieser Index wird sortiert, was mit wesentlich weniger Kopierarbeit verbunden ist, weil die Sätze kürzer sind. Suche ich nun nach einem Datensatz mit einem bestimmten Schlüssel, so gehe ich in den sortierten Index, suche dort möglichst geschickt nach dem Schlüssel und springe dann zu der bei dem Schlüssel stehenden Speicheradresse des Datensatzes. Ich kann auch zwei Indexfiles mit verschiedenen SortierschLüsseln (Namen,

Geburtstag) anlegen, während die Datensätze bestenfalls nach einem einzigen Schlüssel sortiert sein können. Meistens verzichtet man darauf, die Datensätze zu sortieren. Der dritte Band von DONALD E. KNUTH befasst sich nur mit Suchen und Sortieren, ein interessantes Gebiet. Jeder versteht die Aufgaben, die Lösungen können trickreich sein.

Ein weiteres Einsatzgebiet von Pointern ist die Parameterübergabe von Funktionen, siehe Abschnitt 1.4.3 *Parameterübergabe* auf Seite 82. Und schließlich können C-Funktionen nur einen einzigen Wert zurückgeben. Setzt sich ihr Ergebnis aus mehreren Werten zusammen – wie bei einem String oder einem Vektor – so geht das nur über einen Pointer auf das erste Element des Ergebnisses.

Auf Variablen kann mittels ihres Namens oder ihrer Speicheradresse (Hausnummer) zugegriffen werden. Die Speicheradresse braucht nicht absolut oder relativ zu einem Anfangswert bekannt zu sein, sondern ist wiederum über einen Namen ansprechbar, den Namen eines **Pointers**. Genaugenommen gehören die Adressen zur Hardware und sind für den Programmierer fast immer bedeutungslos, während die Pointer Operanden der Programmiersprache sind, denen zur Laufzeit als Wert Adressen zugewiesen werden. Deshalb werden sie auch als **Adressvariable**[26] bezeichnet. Pointer haben Namen, Adressen sind hexadezimale Zahlen. Das Arbeiten mit Adressen beziehungsweise Pointern erlaubt gelegentlich eine elegante Programmierung, ist manchmal unvermeidlich und im übrigen älter als die Verwendung von Variablennamen. Man muss nur stets sorgfältig die Variable von ihrem Pointer unterscheiden. Wenn man Arrays von Pointern auf Strings verwaltet, wird das schnell unübersichtlich. Es ist gute Praxis, aber nicht zwingend, Pointernamen mit einem `p` beginnen oder aufhören zu lassen.

Ein Pointer ist immer ein Pointer auf einen Variablentyp, unter Umständen auf einen weiteren Pointer. Typlose Pointer gibt es nicht in C[27]. Der Wert eines Pointers ist *keine* Ganzzahl (`int`) und darf nicht wie eine Ganzzahl behandelt werden, obwohl letzten Endes die Speicheradressen (Hausnummern) ganze Zahlen sind. Die zulässigen, sinnvollen Operationen sind andere als bei ganzen Zahlen. Hausnummern sind Zahlen, die Multiplikation zweier Hausnummern ist möglich, ergibt jedoch nichts Sinnvolles. Genauso ist es mit Pointern.

Aus einem Variablennamen `x` entsteht der Pointer auf die Variable `&x` durch Voransetzen des **Referenzierungsoperators** `&`. Umgekehrt wird aus dem Pointer `p` die zugehörige Variable `*p` durch Voransetzen des **Dereferenzierungsoperators** `*`. Referenziert werden kann nur ein Objekt im Speicher, also eine Variable, aber nicht ein Ausdruck oder eine Konstante[28]. Dereferenziert werden kann nur ein Pointer, der bereits auf ein Objekt im Speicher verweist, der also eine Speicheradresse enthält. Eine Speicheradresse belegt ein Objekt erst, wenn es definiert ist (einen Wert hat), nicht schon mit der Deklaration. Folgende Zeilen sind zulässig beziehungsweise nicht:

```
int x = 12, *py;

*py = x;    /* zu frueh, unzulaessig  */
py = &x;
*py = x;    /* erlaubt, aber ueberfluessig */

printf("%d  %d\n", x, *py);
```

Wir deklarieren eine Variable `x` als ganzzahlig und weisen ihr zugleich einen Wert zu. Sie ist damit definiert und belegt einen Speicherplatz. Ferner deklarieren wir `py` als Pointer auf eine Ganzzahl. Der erste Versuch, `py` zu dereferenzieren, ist verfrüht und führt zu einem tödlichen Bus Error, da noch kein Objekt `y` definiert ist, dessen Adresse der Pointer `py` enthalten könnte. Der Pointer ist deklariert, aber nicht definiert. Wohl aber kann ich die deklarierte und definierte Variable `x` referenzieren und ihre Adresse dem Pointer `py` zuweisen. Damit

[26]Es gibt natürlich auch Adresskonstanten, deren Wert während des Programmablaufs – von der Initialisierung abgesehen – konstant bleibt.

[27]Der in ANSI-C eingeführte Pointer auf den Typ `void` ist ein Pointer, der zunächst auf keinen bestimmten Typ zeigt.

[28]Konstanten sind Teil des Programmcodes.

enthält auch er einen Wert – und zwar die Adresse von x – und darf beliebig weiterverwendet werden. Ausgegeben wird zweimal der Wert 12. Die Zuweisung des Wertes von `x` an `*py` ist überflüssig, da `py` auf die Adresse zeigt, unter der `x` abgelegt ist. Das Beispiel verdeutlicht den Unterschied zwischen Deklaration und Definition und zeigt, dass man eine Variable – genauso einen Pointer – außer auf der linken Seite einer Zuweisung erst dann verwenden darf, wenn sie einen Wert hat.

Der Name von **Arrays** ist die **Adresskonstante** ihres ersten Elementes (Index 0). Die Bezeichnung des Arraynamens als *Pointer* ist nicht korrekt, aber gebräuchlich. Pointer als (Adress-)Variable können auf der linken Seite einer Zuweisung auftauchen, ein Arrayname ist wie jede Konstante als Linkswert ungeeignet. Der Name von **Funktionen** ohne das Klammernpaar ist die Adresskonstante mit der Einsprungadresse der Funktion, auf die erste ausführbare Anweisung.

Ein Pointer, der auf die Adresse `NULL` verweist, wird **Nullpointer** genannt und zeigt auf kein gültiges Datenobjekt. Wie im richtigen Leben gibt es keine Hausnummer Null. Sein Auftreten kennzeichnet eine Ausnahme oder einen Fehler. Der Wert `NULL` ist der einzige, der direkt einem Pointer zugewiesen werden kann; jede Zuweisung einer Ganzzahl ist ein Fehler, da Pointer keine Ganzzahlen sind. Ansonsten dürfen nur Werte, die sich aus einer Pointeroperation oder einer entsprechenden Funktion (deren Ergebnis ein Pointer ist) einem Pointer zugewiesen werden.

Für Pointer sind die Operationen Inkrementieren, Dekrementieren und Vergleichen zulässig. Die Multiplikation zweier Pointer dürfen Sie versuchen, es kommt aber nichts Brauchbares heraus, meist ein Laufzeitfehler (memory fault). Inkrementieren bedeutet Erhöhung um eine oder mehrere Einheiten des Typs, auf den der Pointer verweist. Dekrementieren entsprechend eine Verminderung. Sie brauchen nicht zu berücksichtigen, um wieviele Bytes es geht, das weiß der Compiler aufgrund der Deklaration. Diese **Pointer-Arithmetik** erleichtert das Programmieren erheblich; in typlosen Sprachen muss man Bytes zählen.

Wir wollen anhand einiger Beispiele mit Arrays den Gebrauch von Pointern verdeutlichen und deklarieren ein eindimensionales Array von vier Ganzzahlen:

```
int a[4];
```

Der Name `a` für sich allein ist der Pointer (Pointerkonstante) auf den Anfang des Arrays. Es sei mit den Zahlen 4, 7, 1 und 2 besetzt. Dann hat es folgenden Aufbau:

Pointer		Speicher	Variable = Wert
a	$\leftrightarrow$	4	$*a = a[0] = 4$
$a + 1$	$\leftrightarrow$	7	$*(a + 1) = a[1] = 7$
$a + 2$	$\leftrightarrow$	1	$*(a + 2) = a[2] = 1$
$a + 3$	$\leftrightarrow$	2	$*(a + 3) = a[3] = 2$

Der Pfeil ist zu lesen als *zeigt auf* oder *ist die Adresse von*. Der Wert des Pointers `a` – die Adresse also, unter der die Zahl 4 abgelegt ist – ist irgendeine kaum verständliche und völlig belanglose Hexadezimalzahl. Der Wert der Variablen `a[0]` hingegen ist 4 und das aus Gründen, die im wirklichen Leben zu suchen sind. Ein Zugriff auf das nicht deklarierte Element $a[4]$ führt spätestens zur Laufzeit auf einen Fehler. Bei der Deklaration des Arrays muss seine Länge bekannt sein. Später, wenn es nur um den Typ geht – wie bei der Parameterübergabe – reicht die Angabe `int *a`.

Ein String ist ein Array von Zeichen (characters), abgeschlossen mit dem unsichtbaren ASCII-Zeichen Nr. 0, hier dargestellt durch $\otimes$. Infolgedessen muss das Array immer ein Element länger sein als der String Zeichen enthält. Wir deklarieren einen ausreichend langen String und belegen ihn gleichzeitig mit dem Wort *UNIX*:

```
char s[6] = "UNIX";
```

Die Längenangabe 6 könnte entfallen, da der Compiler aufgrund der Zuweisung der Stringkonstanten die Länge weiß. Der String ist unnötig lang, aber vielleicht wollen wir später ein anderes Wort darin unterbringen. Das Array sieht dann so aus:

Pointer (Adresse)		Speicher	Wert (Variable)
s	$\leftrightarrow$	U	$*s = s[0] = U$
$s+1$	$\leftrightarrow$	N	$*(s+1) = s[1] = N$
$s+2$	$\leftrightarrow$	I	$*(s+2) = s[2] = I$
$s+3$	$\leftrightarrow$	X	$*(s+3) = s[3] = X$
$s+4$	$\leftrightarrow$	$\otimes$	$*(s+4) = s[4] = \otimes$
$s+5$	$\leftrightarrow$	??	$*(s+5) = s[5] = ??$

Die Fragezeichen deuten an, dass diese Speicherstelle nicht mit einem bestimmten Wert belegt ist. Der Zugriff ist erlaubt; was darin steht, ist nicht abzusehen. Man darf nicht davon ausgehen, dass Strings immer mit Spaces initialisiert werden oder Zahlen mit Null.

Wir deklarieren nun ein zweidimensionales Array von Ganzzahlen, eine nicht-quadratische Matrix:

```
int a[3][4];
```

die mit folgenden Werten belegt sei:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

Im Arbeitsspeicher steht dann folgendes:

Pointer 2.		Pointer 1.		Speicher	Wert (Variable)
a	$\leftrightarrow$	$a[0]$	$\leftrightarrow$	1	$**a = *a[0] = a[0][0] = 1$
	$\leftrightarrow$	$a[0]+1$	$\leftrightarrow$	2	$*(a[0]+1) = a[0][1] = 2$
	$\leftrightarrow$	$a[0]+2$	$\leftrightarrow$	3	$*(a[0]+2) = a[0][2] = 3$
	$\leftrightarrow$	$a[0]+3$	$\leftrightarrow$	4	$*(a[0]+3) = a[0][3] = 4$
$a+1$	$\leftrightarrow$	$a[1]$	$\leftrightarrow$	5	$*a[1] = a[1][0] = 5$
	$\leftrightarrow$	$a[1]+1$	$\leftrightarrow$	6	$*(a[1]+1) = a[1][1] = 6$
	$\leftrightarrow$	$a[1]+2$	$\leftrightarrow$	7	$*(a[1]+2) = a[1][2] = 7$
	$\leftrightarrow$	$a[1]+3$	$\leftrightarrow$	8	$*(a[1]+3) = a[1][3] = 8$
$a+2$	$\leftrightarrow$	$a[2]$	$\leftrightarrow$	9	$*a[2] = a[2][0] = 9$
	$\leftrightarrow$	$a[2]+1$	$\leftrightarrow$	10	$*(a[2]+1) = a[2][1] = 10$
	$\leftrightarrow$	$a[2]+2$	$\leftrightarrow$	11	$*(a[2]+2) = a[2][2] = 11$
	$\leftrightarrow$	$a[2]+3$	$\leftrightarrow$	12	$*(a[2]+3) = a[2][3] = 12$

Der Pointer 2. Ordnung a zeigt auf ein Array aus 3 Pointern 1. Ordnung a[0], a[1] und a[2]. Die Pointer 1. Ordnung a[0], a[1] und a[2] zeigen ihrerseits auf 3 Arrays bestehend aus je 4 Ganzzahlen. Gespeichert sind 12 Ganzzahlen, die Pointer 1. Ordnung sind nicht gespeichert. Da die Elemente allesamt gleich groß sind (gleich viele Bytes lang), hindert uns nichts daran, das Element `a[1][2]`, nämlich die Zahl 7, als Element `a[0][6]` aufzufassen. Die gespeicherten Werte lassen sich auch als eindimensionales Array `b[12]` verstehen. Solche Tricks müssen sorgfältig kommentiert werden, sonst blickt man nach kurzer Zeit nicht mehr durch und Außenstehende nie. Versuchen Sie, an Hand obigen Schemas folgende Behauptungen nachzuvollziehen:

$$**(a+1) = 5(a+1) = a[0]+4 = a[1](*(a+2)-1) = 8(*(a+1)+1) = *(a[1]+1) = 6$$

Im Programm 1.16 `zeit.c` auf Seite 36 haben wir ein Array von Strings kennengelernt, also ein Array von Arrays von Zeichen, abgeschlossen jeweils mit dem ASCII-Zeichen Nr. 0. Es enthält die Namen der Wochentage, mit Spaces aufgefüllt auf gleiche Länge:

```
char *ptag[] = {"Sonntag, ", "Montag,  ", ...};
```

Hier wird ein Array von 7 Pointern gespeichert. Dazu kommen natürlich noch die Strings, die wegen des Aussehens auf dem Bildschirm gleich lang sind, aber vom Programm her ungleich lang sein dürften.

Pointer 2.		Pointer 1.		Sp.	Wert (Variable)
$ptag$	$\leftrightarrow$	$ptag[0]$	$\leftrightarrow$	S	$**ptag = *ptag[0] = ptag[0][0] = S$
	$\leftrightarrow$	$ptag[0]+1$	$\leftrightarrow$	o	$*(ptag[0]+1) = ptag[0][1] = o$
	$\leftrightarrow$	$ptag[0]+2$	$\leftrightarrow$	n	$*(ptag[0]+2) = ptag[0][2] = n$
	$\leftrightarrow$	$ptag[0]+3$	$\leftrightarrow$	n	$*(ptag[0]+3) = ptag[0][3] = n$
	$\leftrightarrow$	$ptag[0]+4$	$\leftrightarrow$	t	$*(ptag[0]+4) = ptag[0][4] = t$
	$\leftrightarrow$	$ptag[0]+5$	$\leftrightarrow$	a	$*(ptag[0]+5) = ptag[0][5] = a$
	$\leftrightarrow$	$ptag[0]+6$	$\leftrightarrow$	g	$*(ptag[0]+6) = ptag[0][6] = g$
	$\leftrightarrow$	$ptag[0]+7$	$\leftrightarrow$	,	$*(ptag[0]+7) = ptag[0][7] =,$
	$\leftrightarrow$	$ptag[0]+8$	$\leftrightarrow$		$*(ptag[0]+8) = ptag[0][8] =$
	$\leftrightarrow$	$ptag[0]+9$	$\leftrightarrow$	$\otimes$	$*(ptag[0]+9) = ptag[0][9] = \otimes$
$ptag+1$	$\leftrightarrow$	$ptag[1]$	$\leftrightarrow$	M	$*ptag[1] = ptag[1][0] = M$
	$\leftrightarrow$	$ptag[1]+1$	$\leftrightarrow$	o	$*(ptag[1]+1) = ptag[1][1] = o$
	$\leftrightarrow$	$ptag[1]+2$	$\leftrightarrow$	n	$*(ptag[1]+2) = ptag[1][2] = n$

Wir brechen nach dem n von Montag ab. ptag ist die Adresse des Speicherplatzes, in dem ptag[0] abgelegt ist. ptag[0] ist die Adresse des Speicherplatzes, in dem das Zeichen S abgelegt ist. Durch zweimaliges Dereferenzieren von ptag erhalten wir das Zeichen S. Die anderen Zeichen liegen auf höheren Speicherplätzen, deren Adressen wir durch Inkrementieren entweder von ptag oder von ptag[ ] erhalten, wobei man normalerweise die zweidimensionale Struktur des Arrays berücksichtigt, obwohl dem Computer das ziemlich gleich ist.

Es sind noch weitere Schreibweisen möglich, die oben wegen der begrenzten Breite nicht unterzubringen sind. Greifen wir die letzte Zeile heraus, das n von Montag. Rein mit Indizes geschrieben gilt:

$$ptag[1][2] = n$$

So entwerfen wir vermutlich ein Programm, weil wir das Arbeiten mit Indizes aus der Mathematik gewohnt sind. Unter Ausnutzen der Pointer-Arithmetik gilt aber auch:

$$*(*(ptag+1)+2) = *(ptag[1]+2) = (*(ptag+1))[2] = ptag[1][2] = n$$

Für den Computer ist Pointer-Schreibweise mit weniger Arbeit verbunden, da er Adressen kennt und Indizes erst in Adressen umrechnen muß.

```
/* array2.c, Indizes und Pointer */

#include <stdio.h>

char a[] = "abcd";    /* Array of chars */

int main()
{
printf("a[2] = %c\n", a[2]);
printf("*(a + 2) = %c\n", *(a + 2));
printf("*(2 + a) = %c\n", *(2 + a));
printf("2[a] = %c\n", 2[a]);
return 0;
}
```

Quelle 1.23 : C-Programm zur Verdeutlichung der Pointerarithmetik

Nun eine leicht verrückte, aber richtige Überlegung. Wir deklarieren und initialisieren einen String `a[]`. Mit Hilfe der Standardfunktion `printf()` geben wir das Element mit der Hausnummer 2 aus, also das Zeichen c. Dann greifen wir auf dasselbe Element zu, indem wir den Pointer auf den Anfang des Arrays um 2 hochzählen und anschließend dereferenzieren.

Jetzt fällt uns ein, dass die Addition – auch in der Pointer-Arithmetik – kommutativ ist, wir vertauschen die beiden Summanden. Erwartungsgemäß geht das gut. Schließlich wandeln wir die Pointer-Schreibweise wieder zurück in die Index-Schreibweise unter Beibehaltung der vertauschten Reihenfolge. Auch das funktioniert, logisch.

Pointer und Variable gehören verschiedenen **Referenzebenen** (level) an, die nicht gemischt werden dürfen. Der gleiche Fall liegt auch in der Linguistik vor, wenn über Wörter gesprochen wird. Vergleichen Sie die beiden sinnvollen Sätze:

- Kaffee ist ein Getränk.
- Kaffee ist ein Substantiv.

Im ersten Satz ist die Flüssigkeit gemeint, im zweiten das Wort, was gelegentlich durch Kursivschreibung oder Gänsefüßchen angedeutet wird. Der erste Satz ist einfaches Deutsch, der zweite gehört einer **Metasprache** an, in der Aussagen über die deutsche Sprache vorgenommen werden, verwirrenderweise mit denselben Wörtern und derselben Grammatik. Genauso kann `x` eine Variable oder ein Pointer auf ein Variable sein oder ein Pointer auf einen Pointer auf eine Variable. Erst ein Blick auf die Deklaration schafft Klarheit.

In C-Programmen wird gern von Pointern Gebrauch gemacht. In der C-Bibel von BRIAN W. KERNIGHAN und DENNIS M. RITCHIE ist ihnen daher ein ganzes, gut lesbares Kapitel gewidmet.

1.3.6.8 Weitere Namen für Typen (typedef)

Mithilfe der `typedef`-Anweisung kann sich der Benutzer eigene, zusätzliche **Namen** für C-Datentypen schaffen. Der neue Name muss eindeutig sein, darf also nicht mit einem bereits anderweitig belegten Namen übereinstimmen. `typedef` erzeugt keinen neuen Datentyp, sondern veranlasst den Compiler, im Programm den neuen Namen wörtlich durch seine Definition zu ersetzen, was man zur Prüfung auch von Hand machen kann. Der neue Typname ist ein Synonym. Der Zweck neuer Typnamen ist eine Verbesserung der Lesbarkeit und Portierbarkeit des Quelltextes.

Einige Beispiele. Wir wollen uns einen Typnamen `BOOLEAN` schaffen, der zwar im Grunde nichts anderes ist als der Typ `int`, aber die Verwendung deutlicher erkennen lässt. Zu Beginn der Deklarationen oder vor `main()` schreiben wir:

```
typedef   int   BOOLEAN;
```

(die Großschreibung ist nicht zwingend) und können anschließend eine Variable `janein` als `BOOLEAN` deklarieren

```
BOOLEAN    janein;
```

Der Compiler ersetzt den String `BOOLEAN` in Deklarationen durch den String `int`, ähnlich wie es der Präprozessor bei `#define`-Anweisungen macht.

In FORTRAN gibt es den Datentyp `complex`, den wir in C durch eine Struktur nachbilden:

```
typedef struct {
    double real;
    double imag;
} COMPLEX;
```

Hiermit ist nur ein neuer, einfacherer Name für den Strukturtyp `struct` geschaffen worden. Dann deklarieren wir die komplexe Variable:

```
COMPLEX z, R[20];
```

`z` ist eine komplexe Variable, `R` ein Array von 20 komplexen Variablen. Leider ist damit noch nicht alles erledigt, denn die arithmetischen Operatoren von C gelten nur für Ganz- und Gleitkommazahlen, nicht für Strukturen. Wir müssen noch Funktionen für die Operationen

mit komplexen Variablen schreiben. In FORTRAN hingegen gelten die gewohnten arithmetischen Operatoren auch für komplexe Daten. In C++ lassen sich die Bedeutungen der Operatoren erweitern (überladen), aber in C nicht.

Bei Strings taucht das Problem der Längenangabe auf. Folgender Weg ist gangbar, erfüllt aber nicht alle Wünsche:

```
typedef   char   *STRING;
```

Dann können wir schreiben

```
STRING fehler = "Falsche Eingabe";
```

Der Compiler weiß die Länge der Strings aufgrund der Zuweisung der Stringkonstanten. Hingegen ist die nachstehende Deklaration fehlerhaft, wie man durch Einsetzen erkennt:

```
STRING   abc[16];
```

Die Typdefinition eingesetzt ergibt:

```
char   *abc[16];
```

und das ist kein String, sondern ein Array von Strings. Erst zweimaliges Dereferenzieren führt auf den Typ `char`. Die Schreibweise:

```
typedef   char   [16]   STRING;
STRING abc;
```

die dieses Problem lösen würde, haben wir zwar in einem Buch gefunden, wurde aber nicht von unserem Compiler angenommen.

Ist man darauf angewiesen, dass ein Datentyp eine bestimmte Anzahl von Bytes umfasst, erleichtert man das Portieren, indem man einen eigenen Typnamen deklariert und im weiteren Verlauf nur diesen verwendet. Bei einer Portierung ist dann nur die Typdefinition anzupassen. Es werde eine Ganzzahl von vier Byte Länge verlangt. Dann deklariert man:

```
typedef   int   INT4;   /* Ganzzahl von 4 Bytes */
INT4   i, j, k;
```

und ändert bei Bedarf nur die `typedef`-Zeile. Zweckmäßig packt man die Typendefinition in eine private include-Datei, die man für mehrere Programme verwenden kann.

1.3.6.9 Speicherklassen

In C gibt es vier **Speicherklassen** (storage classes):

- auto
- extern
- register
- static

Die Speicherklasse geht dem Typ in der Deklaration voraus:

```
static int x;
```

Die Klasse `auto` ist die Defaultklasse für **lokale Variable** und braucht nicht eigens angegeben zu werden. Variablen dieser Klasse leben nur innerhalb des Bereiches, in dem sie deklariert wurden, und sterben beim Verlassen des Bereiches, der von ihnen belegte Speicher wird freigegeben.

Eine **globale Variable** darf in einem Programm nur einmal deklariert werden. Erstreckt sich ein Programm über mehrere getrennt zu kompilierende Dateien, so darf sie nur in einer der Dateien deklariert werden. Da aber der Compiler auch in den übrigen Dateien den Typ der Variablen kennen muss, wird die Variable hier als **extern** deklariert. Globale Variable sind per Default extern. Funktionen gehören stets der Speicherklasse extern an.

register-Variable werden nach Möglichkeit in Registern nahe dem Rechenwerk gehalten und sind damit schnell verfügbar. Ansonsten verhalten sie sich wie auto-Variable. Sie müssen vom Typ int oder char sein. Eine typische Anwendung sind Schleifenzähler. Optimierende Compiler ordnen von sich aus einige Variable dieser Speicherklasse zu. Auf `register`-Variable kann der Referenzierungs-Operator `&` nicht angewendet werden. Es ist auch unsicher, ob das System der `register`-Anweisung folgt. Am besten verzichtet man auf diese Speicherklasse.

Lokale Operanden gelten und leben nur innerhalb des Blockes, in dem sie deklariert wurden. Durch die Zuordnung zur Speicherklasse **static** verlängert man ihre **Lebensdauer** – nicht ihren Geltungsbereich – über das Ende des Blockes hinaus. Bei einem erneuten Aufruf des Blockes hat ein static-Operand den Wert, den er beim vorherigen Verlassen des Blockes hatte.

1.3.6.10 Geltungsbereich

Eine Variable gilt nur innerhalb des Bereiches (Programmabschnittes), zu dessen Beginn[29] sie deklariert worden ist. Ihr Sichtbarkeits- oder **Geltungsbereich** (scope) ist dieser Bereich. Außerhalb des zugehörigen Bereiches ist die Variable unbekannt (nicht existent) oder unsichtbar (existent, aber nicht zugänglich). Der Name der Variablen ist in diesem Zusammenhang bedeutungslos. Ein Bereich ist:

- ein Programm,
- eine Funktion,
- ein logischer Block zwischen `{` und `}`,
- eine Datei.

Variable, die vor allen Funktionen – in der Regel vor `main()` – deklariert werden, gelten infolgedessen global in allen Funktionen, die in derselben Datei deklariert werden, das heißt im ganzen Programm, wenn dieses nur aus einer Datei besteht. Variable, die zu Beginn einer Funktion – vor allen Anweisungen in der Funktion – deklariert werden, gelten innerhalb dieser Funktion, aber nicht außerhalb. Sie sind lokal gültig. Variable, die zu Beginn eines Blockes – vor allen Anweisungen im Block – deklariert werden, gelten nur in diesem Block. Das kommt selten vor, ist aber völlig in Ordnung.

Erstreckt sich ein Programm über mehrere Dateien, so gelten zu Beginn einer Datei – vor den darin enthaltenen Funktionen – deklarierte Operanden global für die Funktionen in der Datei, aber nicht für das gesamte Programm. Soll ein Operand global im gesamten Programm gelten, so ist er in jeder Datei erneut zu deklarieren. Dies widerspricht jedoch der Forderung, dass ein- und derselbe Operand nur einmal deklariert werden darf. Der Ausweg liegt darin, ihn nur einmal in beschriebener Weise zu deklarieren und in allen anderen Dateien vor die Deklaration das Wort `extern` zu setzen. Damit ist der Forderung nach Eindeutigkeit Genüge getan, und der Compiler weiß trotzdem bei jeder Datei, mit welchen Typen er es zu tun hat.

Wird ein Operand desselben Namens innerhalb eines Bereiches nochmals deklariert, so hat für diesen Bereich die lokale Deklaration Vorrang vor der äußeren Deklaration. Es wird ein neuer, lokaler Operand erzeugt. Der Geltungsbereich des äußeren Operanden hat eine Lücke, der äußere Operand wird verschattet.

Das Konzept des Geltungsbereiches lässt sich über ein Programm hinaus erweitern. Die exportierten Umgebungs-Variablen der Sitzungsshell gelten für alle Prozesse einer Sitzung und können von diesen abgefragt oder verändert werden. Darüber hinaus sind auch Variable denkbar, die in einem Verzeichnis, einem Datei-System oder in einer Netz-Domain gelten. Je größer der Geltungsbereich ist, desto sorgfältiger muss man mit der Schreibberechtigung umgehen.

---

[29] In C++ dürfen Variable an beliebiger Stelle innerhalb eines Bereiches deklariert werden, jedoch muss dies vor ihrer ersten Verwendung erfolgen.

1.3.6.11 Lebensdauer

Beim Eintritt in einen Bereich wird für die in diesem Bereich definierten Variablen Speicher zugewiesen (allokiert). Beim Verlassen des Bereiches wird der Speicher freigegeben, von den Variablen bleibt keine Spur zurück. Ihre **Lebensdauer** (lifetime) ist die aktive Zeitspanne des Bereiches. Beim nächsten Aufruf des Bereiches wird neuer Speicher zugewiesen und initialisiert. Diese Speicherklasse wird als `auto` bezeichnet und ist die Standardklasse aller Variablen, für die nichts anderes vereinbart wird.

Möchte man jedoch mit den alten Werten weiterrechnen, so muss man die Variable der Speicherklasse `static` zuweisen. Der Geltungsbereich wird davon nicht berührt, aber der Speicher samt Inhalt bleibt beim Verlassen der Funktion bestehen. Die Variable besteht, ist aber vorübergehend unsichtbar (existent, aber nicht zugänglich).

1.3.7 Operationen

1.3.7.1 Ausdrücke

Wir haben bisher Operanden betrachtet, aber nichts mit ihnen gemacht. Nun wollen wir uns ansehen, was man mit den Operanden anstellen kann. Der **Operator** bestimmt, was mit dem Operand geschieht. Unäre Operatoren wirken auf genau einen Operanden, binäre auf zwei, ternäre auf drei. Mehr Operanden sind selten. Operator plus Operanden bezeichnet man als **Ausdruck** (expression). Ein Ausdruck hat nach seiner Auswertung einen **Wert** und kann überall dort stehen, wo ein Wert verlangt wird. Eine Funktion, die einen Wert zurückgibt, kann anstelle eines Ausdrucks oder Wertes stehen.

1.3.7.2 Zuweisung

Eine **Zuweisung** (assignment) weist einer Variablen einen Wert zu. Der Operator ist das Gleichheitszeichen (ohne Doppelpunkt wie in PASCAL, wegen Faulheit). Das Gleichheitszeichen darf von Spaces umgeben sein und sollte es wegen der besseren Lesbarkeit auch. Wert und Variable sollten vom selben Typ sein. Es gibt zwar in C automatische Typumwandlungen, aber man sollte wenig Gebrauch davon machen. Sie führen zu den berüchtigten unlesbaren C-Programmen und gefährden die Portabilität.

Da ein Ausdruck wie eine Summe oder eine entsprechende Funktion einen Wert abliefert, kann in einer Zuweisung anstelle des Wertes immer ein Ausdruck stehen. Die Zuweisung selbst liefert den zugewiesenen Wert zurück und kann daher als Wert in einem übergeordneten Ausdruck auftreten.

Auf der rechten Seite einer Zuweisung kann alles stehen, was einen Wert hat, beispielsweise eine Konstante, ein berechenbarer Ausdruck oder eine Funktion, aber kein Array und damit auch kein String. Solche Glieder werden als **r-Werte** (r-value) bezeichnet. Auf der linken Seite einer Zuweisung kann alles stehen, was einen Wert annehmen kann, beispielsweise eine Variable, aber keine Konstante und keine Funktion. Diese Glieder heißen **l-Werte** (l-value).

Eine Zuweisung ist *keine* mathematische Gleichung. Die Formel

$$x = x + 1 \tag{1.1}$$

ist als Gleichung für jeden endlichen Wert von x falsch, als Zuweisung dagegen ist sie gebräuchlich und bedeutet: Addiere 1 zu dem Wert von x und schreibe das Ergebnis in die Speicherstelle von x. Damit erhält die Variable x einen neuen Wert. Bei einer Gleichung gibt es weder alt noch neu, die Zeit spielt keine Rolle. Bei einer Zuweisung gibt es ein Vorher und Nachher. Die Formel

$$x + 2 = 5 \tag{1.2}$$

hingegen ist als Gleichung in Ordnung, nicht aber als Zuweisung, da auf der linken Seite ein Ausdruck steht und nicht ein einfacher Variablenname. Wegen dieser Diskrepanz zwischen Gleichung und Zuweisung ist letztere etwas umstritten. Ihre Begründung kommt nicht aus der Problemstellung, sondern aus der Hardware, nämlich der Speicherbehandlung. Und gerade diese möchte man mit den höheren Programmiersprachen verdecken.

1.3.7.3 Arithmetische Operationen

Die **arithmetischen Operationen** sind:

- Vorzeichenumkehr - (unärer Operator)
- Addition +
- Subtraktion - (binärer Operator)
- Multiplikation *
- Division /
- Modulus % (Divisionsrest, nur für ganze Zahlen)
- Inkrement ++
- Dekrement --

Inkrement und Dekrement sind Abkürzungen. Der Operator kann vor oder nach dem Operanden (Präfix, Postfix) stehen. Der Ausdruck

```
i++
```

gibt den Wert von `i` zurück und erhöht ihn dann um eins. Der Ausdruck

```
++i
```

erhöht den Wert von `i` um eins und gibt dann den erhöhten Wert zurück. Für das Dekrement gilt das Entsprechende. Der kompilierte Code ist effektiver als der des äquivalenten Ausdrucks

```
i = i + 1
```

Spielt es vom Programm her keine Rolle, ob man den Präfix oder den Postfix verwendet, ist der Präfix `++i` unter Umständen schneller und daher zu bevorzugen. Ferner gibt es noch eine abgekürzte Schreibweise für häufig wiederkehrende Operationen:

```
+=, -=, *=, /=
```

Der Ausdruck

```
y += x
```

weist die Summe der beiden Operanden dem linken Operanden zu und ist somit gleichbedeutend mit dem Ausdruck:

```
y = y + x
```

Entsprechendes gilt für die anderen Abkürzungen. Die ausführliche Schreibweise bleibt weiterhin erlaubt.

Die Division für ganze Zahlen ist eine andere Operation als für Gleitkommazahlen (die übrigen Operationen auch, nur fällt es da nicht auf). In manchen Programmiersprachen werden folgerichtig unterschiedliche Operatoren verwendet, in C nicht. Der Compiler entnimmt aus dem Zusammenhang, welche Division gemeint ist. Diese Mehrfachverwendung eines Operators wird **Überladung** genannt und spielt in objektorientierten Sprachen wie C++ eine Rolle. In FORTRAN, das den komplexen Zahlentyp kennt, gelten die arithmetischen Operatoren auch für diesen. Vorstellbar ist ebenso eine Addition (Verkettung) von Strings.

1.3.7.4 Logische Operationen

Die **logischen Operationen** sind:

- bitweise Negation (not) ~ (Tilde)
- ausdrucksweise Negation (not) `!`
- bitweises Und (and) `&`
- ausdrucksweises Und (and) `&&`
- bitweises exklusives Oder (xor) ^ (Circumflex, caret)
- bitweises inklusives Oder (or) `|`
- ausdrucksweises inklusives Oder (or) `||`

Auch hier gibt es abgekürzte Schreibweisen:

```
^=, |=, &=
```

Der Ausdruck

```
y &= x
```

bedeutet dasselbe wie

```
y = y & x
```

Die Operanden y und x werden bitweise durch Und verknüpft, das Ergebnis wird y zugewiesen. Entsprechendes gilt für die beiden anderen Abkürzungen.

Der Unterschied zwischen einer **ausdrucksweisen** und einer **bitweisen** logischen Operation ist folgender: In C gilt der Zahlenwert 0 als logisch falsch (false), jeder Wert ungleich 0 als logisch wahr (true). Die Zeilen:

```
...
int x = 0;
if (x) printf("if-Zweig\n");
else printf("else-Zweig\n");
...
```

führen zur Ausführung des `else`-Zweiges. Die Variable `x` hat den Wert 0, bei dem auch alle Bits auf 0 stehen. Sowie ein beliebiges Bit auf 1 stünde, hätte die Variable einen Wert ungleich 0 und würde als wahr angesehen. Die ausdrucksweise Negation:

```
if (!x) printf ...
```

kehrt die Verhältnisse um, der `if`-Zweig wird ausgeführt. Die bitweise Negation hätte hier zwar denselben Erfolg, wäre jedoch nicht sinnvoll, da die einzelnen Bits nicht interessieren.

Der Buchstabe G wird in 7-bit-ASCII durch die Bitfolge 1000111 dargestellt. Ihre bitweise Negation ist 0111000, was der Ziffer 8 entspricht. Das Mini-Programm:

```
/* Bitweise Negation */

#include <stdio.h>

int main()
{
char x = 'G';
printf("%c  %c\n", x, ~x);
return 0;
}
```

Quelle 1.24 : C-Programm zur Veranschaulichung der bitweisen Negation

gibt den Buchstaben G und die Ziffer 8 aus, zumindest auf Maschinen, die den 7-bit-ASCII-Zeichensatz verwenden. Der Zweck der bitweisen Operation ist der Umgang mit Informationen, die nicht in einem Wert als Ganzem, sondern in einzelnen Bits stecken. Das kommt bei der Systemprogrammierung vor. Beispielsweise lässt sich die Information, ob ein Gerät ein- oder ausgeschaltet ist, in einem Bit unterbringen. In der Anwendungsprogrammierung ist man meist großzügiger und spendiert eine ganze Integer-Variable dafür.

Merke: Ausdrucksweise logische Operationen und die noch folgenden Vergleichs-Operationen haben ein Ergebnis, das wahr oder falsch lautet (nicht-0 oder 0). Bitweise logische Operationen können jedes beliebige Ergebnis im Bereich der ganzen Zahlen haben.

1.3.7.5 Vergleiche

Die **Vergleichs-** oder **Relations-Operationen** sind:

- gleich == (weil = schon die Zuweisung ist)
- ungleich `!=`
- kleiner <
- kleiner gleich <=
- größer >
- größer gleich >=
- Bedingte Bewertung `?:`

Das Ergebnis eines Vergleichs ist ein boolescher Wert, also `true` oder `false` beziehungsweise in C die entsprechenden Zahlen nicht-0 oder 0.

Ein häufiger Fehler ist die Verwendung des einfachen Gleichheitszeichens für die Abfrage auf Gleichheit. Dieser Fehler ist schwierig zu erkennen, da der fehlerhafte Ausdruck syntaktisch korrekt ist, er bedeutet nur eine Zuweisung an Stelle des beabsichtigten Vergleichs:

```
if (x = 0) { ...    /* statt (x == 0) */
```

Der Compiler protestiert nicht. Da in Vergleichen oft auf einer Seite Ausdrücke wie Konstanten vorkommen, die nicht auf der linken Seite einer Zuweisung stehen können (r-values), empfiehlt es sich, diese auf die linke Seite des Vergleichs zu stellen:

```
if (0 == x) { ..
```

Bei dem falschen Operator protestiert dann der Compiler oder Syntaxprüfer. Einfach eine kleine Angewohnheit, die die Arbeit erleichtert.

Pfiffig ist die **Bedingte Bewertung** (conditional operator), auch `Bedingter Ausdruck` genannt. Der Ausdruck:

```
z = (a < 0) ? -a : a
```

hat dieselbe Wirkung wie:

```
if (a < 0)
    z = -a;
else
    z = a;
```

Er weist der Variablen `z` den Betrag von `a` zu. Rechts des Gleichheitszeichens stehen drei Ausdrücke, die auch zusammengesetzt sein dürfen. Die ganze Bedingte Bewertung ist selbst wieder ein Ausdruck wie eine Zuweisung und kann überall stehen, wo ein Wert verlangt wird. Die Bedingte Bewertung ist einer der seltenen ternären oder triadischen Operatoren (drei Operanden) und führt zu schnellerem Code als `if - else`. Welchen Wert nimmt `z` in folgendem Beispiel an?

```
z = (a >= b) ? a : b
```

Eine Anwendung finden Sie im Abschnitt 1.4.7 *Rekursiver Aufruf einer Funktion* auf Seite 96.

1.3.7.6 Bitoperationen

Die **Bit-Operationen** sind:

- shift links <<
- shift rechts >>

Bei vorzeichenlosen Ganzzahlen ist ein Shiften nach links gleichbedeutend mit einer Multiplikation mit 2, ein Shiften nach rechts mit einer Division durch 2. Auf die links oder rechts wegfallende Stelle ist zu achten, nachgeschoben am anderen Ende wird eine Null. Die Shift-Operation ist schnell. Weiterhin beziehen sich einige logische Operationen auf Bits (siehe oben). Auch hier sind Abkürzungen möglich:

`<<=, >>=`

Der Ausdruck

`y <<= x`

ist gleichbedeutend mit

`y = y << x`

Der linke Operand wird um so viele Bits nach links verschoben, wie der rechte Operand angibt. Das Ergebnis wird dem linken Operanden zugewiesen. Zur Veranschaulichung der Bitoperationen ein kleines Programm:

```
/* Programm mit Bitoperationen,
   sinnvolle Argumente z. B. 8  2 */

#include <stdio.h>
void exit();

int main(int argc,char *argv[])
{
int i, j, k;

if (argc < 3) {
    puts("Zwei Argumente erforderlich.");
    exit(-1);
}
sscanf(argv[1], "%d", &i);
sscanf(argv[2], "%d", &j);

k = i « j;
printf("Eingabe %d um %d Bits nach links:  %d\n", i, j, k);
k = i » j;
printf("Eingabe %d um %d Bits nach rechts: %d\n", i, j, k);
k = i & j;
printf("Eingabe %d mit %d bitweise und:   %d\n", i, j, k);
k = i | j;
printf("Eingabe %d mit %d bitweise oder:   %d\n", i, j, k);

return 0;
}
```

Quelle 1.25 : C-Programm mit Bitoperationen

1.3.7.7 st ointeroperationen

Folgende Operationen behandeln Pointer:

- Referenzierung `&`
- Dereferenzierung `*` oder bei Arrays `[ ]`

- Strukturverweis -> (minus größer, bei Strukturpointern)
- Strukturverweis . (Punkt, bei Strukturnamen)

Weiterhin sind folgende für Ganzzahlen zulässige Operationen für Pointer definiert:

- Vergleich zweier Pointer auf denselben Typ
- Inkrementierung (Addition einer ganzen Zahl)
- Dekrementierung (Subtraktion einer ganzen Zahl)
- Subtraktion zweier Pointer desselben Typs

Der Vergleich zweier gleichartiger Pointer auf Übereinstimmung ist immer möglich, ein Vergleich größer-kleiner setzt eine bestimmte Ordnung der Adressen voraus und ist problematisch, man lässt besser die Finger davon:

```
double *p1, *p2;
if (p1 == p2) {}       /* ok */
if (p1 != p2) {}       /* ok */
if (p1 == NULL) {}     /* ok */
if (p1 < p2) {}        /* gefaehrlich */
```

Bei der Addition oder Subtraktion einer ganzen Zahl bedeutet die ganze Zahl *nicht* eine Anzahl von Bytes, sondern eine Anzahl von Objekten des zum Pointer gehörigen Datentyps. Man braucht sich also nicht darum zu kümmern, wieviele Bytes der Datentyp belegt:

```
double x, vektor[10];
x = *vektor;           /* erstes Element, Index 0 */
x = *(vektor + 2);     /* uebernaechstes Element */
```

Die selten vorkommende Subtraktion zweier gleichartiger Pointer liefert die Anzahl der zwischen den beiden Pointern liegenden Datenobjekte.

1.3.7.8 Ein- und Ausgabe-Operationen

In C gibt es keine Operatoren zur Ein- oder Ausgabe, vergleichbar mit `read` oder `write` in PASCAL oder FORTRAN. Stattdessen greift man entweder auf **Systemaufrufe** des Betriebssystems (z. B. UNIX) zurück oder besser auf Funktionen der **C-Standardbibliothek**, die letzten Endes auch Systemaufrufe verwenden, nur intelligent verpackt. Die Systemaufrufe haben eine etwas schwierigere Syntax, erlauben dafür aber auch Dinge außerhalb des Üblichen. Wer portabel für verschiedene Betriebssysteme programmieren möchte, bevorzugt die Standardfunktionen, da sie die Unterschiede verdecken. Wenn keine Gründe dagegen sprechen, nimmt man die Standardfunktionen.

Die UNIX-Systemaufrufe sind in der Sektion 2 des Handbuchs zu finden, die wichtigsten lauten `open(2)`, `close(2)`, `read(2)` und `write(2)`. Hier ein Programmbeispiel:

```
/* Demo Systemaufruf open(2) */

#include <stdio.h>     /* wegen puts(3) */
#include <fcntl.h>     /* wegen open(2) */
#include <unistd.h     /* wegen write(2) */
#include <string.h     /* wegen strlen(3) */

int main(int argc, char *argv[])
{
int fildes;
size_t bufsize;
ssize_t n;
char *buffer = "UNIX ist prima.";

if (argc < 2){
        puts("Filenamen vergessen");
        return(-1);
```

```
}

/* File muss bereits existieren */

fildes = open(argv[1], O_WRONLY);  /* File-Deskriptor */
bufsize = (size_t)strlen(buffer);

if (fildes > 2) {
        puts("open() erfolgreich");
        n = write(fildes, (void *)buffer, bufsize);
        if (n == bufsize) {
                puts("write() erfolgreich");
        }
        close(fildes);
}
else {
        puts("Fehler bei open()");
        return(-1);
}

return 0;
}
```

Quelle 1.26 : C-Programm Ausgabe per Systemaufruf write(2)

Mittels `open(2)` öffnen wir die Datei, deren Name als erstes Argument übergeben wird, zum Schreiben. Die DAtei muss bereits vorhanden sein. Der Systemaufruf gibt einen Datei-Deskriptor zurück, eine fortlaufende Nummer der vom Programm geöffneten Dateien, beginnend mit 3. Dann schreiben wir den in einem Puffer abgelegten String zum Datei-Deskriptor und schließen die Datei. Der Rest sind kleine Maßnahmen zur Fehlerbehandlung.

Die C-Standardfunktionen sind in der Sektion 3 des Handbuchs zu finden, die wichtigsten lauten `fopen(3)`, `fclose(3)`, `scanf(3)` und `printf(3)`. Hier ein Programmbeispiel:

```
/* Demo Standardfunktion fopen(3) */

#include <stdio.h>    /* wegen fopen(3), fputs(3) usw. */

int main(int argc, char *argv[])
{
FILE *fp;             /* File-Pointer */
char *string = "UNIX ist prima.";
int x;

if (argc < 2){
        puts("Filenamen vergessen");
        return(-1);
}

/* File braucht noch nicht zu existieren */

fp = fopen(argv[1], "w");

if (fp != NULL) {
        puts("fopen() erfolgreich");
        x = fputs(string, fp);
        if (x != EOF) {
                puts("fputs() erfolgreich");
        }
        fclose(fp);
}
else {
        puts("Fehler bei fopen()");
        return(-1);
}
```

```
return 0;
}
```

Quelle 1.27 : C-Programm Ausgabe per Standardfunktion fputs(3)

Das Programm macht im Grunde das Gleiche wie das vorangegangene, die Schreibfunktion `fputs(3)` ist jedoch optimiert für das Schreiben von Strings in eine Datei. Es gibt auch eine Standardfunktion `fwrite(3)` zum Schreiben eines Blocks binärer Daten aus einem Puffer in eine Datei. Ein DAtei-Pointer ist ein Pointer auf eine Struktur des Typs `FILE`, die in der Include-Datei `stdio.h` definiert ist.

1.3.7.9 Sonstige Operationen

Ferner bietet C noch einige Operatoren für verschiedene Aufgaben:

- Datentyp-Umwandlung (cast-Operator) `()`
- Komma-Operator `,`
- sizeof-Operator `sizeof()`

Der **cast-Operator** oder Umwandlungsoperator enthält in den runden Klammern eine skalare Typbezeichnung ohne Speicherklasse und geht dem umzuwandelnden skalaren Operanden unmittelbar voraus:

```
int n;
double y;
y = sqrt((double) n);
```

`n` sei eine Ganzzahl. Die Funktion `sqrt()` erwartet jedoch als Argument eine Gleitkommazahl doppelter Genauigkeit. Die Typumwandlung von `n` wird durch den cast-Operator `(double)` bewirkt. Bei der Typumwandlung können Bits oder Dezimalstellen verlorengehen, wenn der neue Typ kürzer ist als der alte. Die Typumwandlung gilt nur im Zusammenhang mit dem cast-Operator und hat für die weiteren Anwendungen der Variablen keine Bedeutung. Manche Compiler beanstanden einen cast-Operator auf der linken Seite einer Zuweisung (als l-Wert).

Der **Komma-Operator** trennt in einer Auflistung von Ausdrücken diese voneinander. Sie werden von links nach rechts abgearbeitet. Das Ergebnis der Auflistung hat Typ und Wert des rechts außen stehenden Ausdrucks. Ein Beispiel:

```
int i, j = 10;
i = (j++, j += 100, 999);
printf("%d", i);
```

Hier wird zuerst `j` um 1 erhöht, dann dazu 100 addiert und schließlich dem gesamten Ausdruck der Wert 999 zugewiesen. `j` hat also nach den Operationen den Wert 111, `i` den Wert 999, der ausgegeben wird. Der Komma-Operator wird oft im Kopf von `for`-Schleifen verwendet. Das Komma zwischen Variablennamen in Deklarationen oder in einer Argumentliste ist kein Operator. Die Reihenfolge der Abarbeitung solcher Listen ist unsicher.

Der **sizeof-Operator** gibt die Größe des Operanden in Bytes zurück. Der Operand kann eine Variable, ein Ausdruck oder ein Datentyp sein, auch ein zusammengesetzter. Ein Ausdruck wird dabei nicht ausgewertet. Dagegen können Funktionen oder Bitfelder nicht als Operand von `sizeof` auftreten. Mit Hilfe des Operators vermeidet man Annahmen über die Größe von Variablen bei Speicherreservierungen, das Programm wird portabler. Der Ausdruck

```
sizeof(x)
```

liefert die Größe von `x` in Bytes als vorzeichenlose Ganzzahl zurück. `sizeof` wird während der Übersetzung ausgewertet (nicht jedoch vom Präprozessor) und verhält sich zur Laufzeit wie eine konstante ganze Zahl.

1.3.7.10 Vorrang und Reihenfolge

Es gibt in C/C++ ähnlich wie in der Mathematik genaue Regeln über den **Vorrang** (precedence) der Operatoren. Es ist jedoch nicht sicher, dass alle C-Compiler sich genau an die Vorgaben des ANSI-Vorschlags halten. Zudem hat man beim Programmieren meist nicht alle Regeln im Kopf, so dass es besser ist, die Ausdrücke durch runde **Klammern** klar und eindeutig zu kennzeichnen. Überflüssige Klammerpaare stören den Compiler nicht. Hier die Liste der Operatoren von C mit nach unten abnehmendem Rang:

```
( )   [ ]   -> .
!   ~   ++   --   -   (cast)   *   &   sizeof
*   /   %
+   -
<<   >>
<   <=   >   >=
==   !=
&
^
|
&&
||
?   :
=   +=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
,
```

Zuerst werden also die runden Klammern ausgewertet; wie in der Mathematik haben sie Vorrang vor allen anderen Operatoren außer den drei weiteren in derselben Zeile. Am schwächsten bindet der Komma-Operator.

Neben ihrem Rang haben Operatoren eine **Assoziativität** oder Leserichtung (associativity). Die Auswertung eines Ausdrucks wie:

```
a + b + c
```

ist durch Vorrang nicht eindeutig zu klären, da die beiden Pluszeichen denselben Rang haben. Dieser Fall tritt immer auf, wenn auf beiden Seiten eines Operanden Operatoren desselben Ranges stehen. Durch den Compiler – nicht durch die Syntax der Sprache, die lässt die Frage offen – ist nun festgelegt, dass die arithmetischen Operatoren links-assoziativ sind. Der Operand in der Mitte wird vom linken Operator geschnappt, so dass die Summe wie folgt ausgewertet wird:

```
(a + b) + c
```

Man könnte auch sagen, dass der Ausdruck von links nach rechts gelesen wird. Eine Zuweisung dagegen ist von rechts nach links zu lesen, sie ist rechts-assoziativ. Zuerst wird die rechte Seite der Zuweisung ausgewertet und dann das Ergebnis der linken Seite zugewiesen. Im Beispiel:

```
a = b = c
```

wird der Wert `c` der Variablen `b` zugewiesen und dann das Ergebnis dieser Zuweisung (der Wert `b`) der Variablen `a`. Die Reihenfolge einer Auswertung wird also zuerst durch den Rang und dann durch die Assoziativität der Operatoren bestimmt. Im Anhang in Abschnitt D.2 *Operatoren* auf Seite 200 sind alle Operatoren von C/C++ mit Rang und Assoziativität aufgelistet.

Bei Funktionsaufrufen ist ungewiss, in welcher Reihenfolge etwaige Argumente ausgewertet werden. Im Beispiel:

```
int n = 5;
printf("%d %d\n", ++n, n * n);
```

ist unsicher, ob n erst inkrementiert und dann quadriert wird oder umgekehrt. Das Ergebnis ist entweder 6, 25 oder 6, 36. Die Argumente von printf() arbeitet der eine Compiler von rechts ab, der andere von links. Nur durch eine eindeutige Schreibweise:

```
int n = 5;
++n;
printf{"%d %d\n", n, n * n);
```

lässt sich ein eindeutiges Ergebnis erreichen – hier 6, 36.

In der Mathematik gibt es nur eine Leserichtung, nämlich von links nach rechts. Über die Reihenfolge der Rechenschritte besagt die Leserichtung nichts. Die obige mehrfache Zuweisung wäre als Gleichung unzulässig, bei der Addition spielt die Reihenfolge keine Rolle für das Ergebnis.

Mißachtung von Rang und Assoziativität führt zu schwierig aufzudeckenden logischen Fehlern im Programm. Syntaktisch ist das Programm richtig, es tut nur etwas anderes, als sich der Programmierer vorgestellt hat. Deshalb ist dringend anzuraten, die Reihenfolge einer Auswertung durch Klammern oder Einzelanweisungen zwingend vorzuschreiben und Ausdrücke zu vermeiden, deren Wert von Vermutungen über die Reihenfolge der Auswertung abhängt.

1.3.8 Anweisungen

1.3.8.1 Leere Anweisung

Anweisungen (statement) haben eine **Wirkung**, aber keinen Wert, im Gegensatz zu Ausdrücken. Die einfachste Anweisung ist die **leere Anweisung**, also die Aufforderung an den Computer, nichts zu tun. Das sieht zwar auf den ersten Blick schwachsinnig aus, ist aber gelegentlich nützlich. Da in C jede Anweisung mit einem Semikolon abgeschlossen werden muss, ist das nackte Semikolon die leere Anweisung. In anderen Sprachen findet sich dafür die Anweisung nop oder noop (no operation). Ein Beispiel:

```
while ((c = getchar()) != 125);
```

Die Schleife liest Zeichen ein und verwirft sie, bis sie auf ein Zeichen Nr. 125 (rechte geschweifte Klammer) trifft. Das wird auch noch entsorgt, dann geht es nach der Schleife weiter.

1.3.8.2 Zuweisung als Anweisung

Aus einer **Zuweisung** wird durch Anhängen eines Semikolons eine Anweisung. Kommt eine Zuweisung beispielsweise als Argument einer Funktion oder in einer Bedingung vor, darf sie nicht durch ein eigenes Semikolon abgeschlossen werden. Die Zuweisung wird ausgeführt und ihr Wert an ihre Stelle gesetzt. Steht die Zuweisung allein, muss sie mit einem Semikolon beendet werden und wird damit zu einer Anweisung an den Computer, etwas zu tun:

```
printf("%d %f \n", x = 3, log(4));
x = 5;
y = log(4);
```

Ähnlich wie die Return-Taste in der Kommandozeile bewirkt hier erst das Semikolon, dass etwas geschieht.

1.3.8.3 Kontrollanweisungen

Kontrollanweisungen steuern in Abhängigkeit von dem Wert eines booleschen Ausdrucks (Bedingung) die Abarbeitung von Programmteilen (einzelnen Anweisungen oder Blöcken), weshalb die Einrichtung auch Ablaufkontrolle genannt wird. Die Kontrollanweisungen von C wie von vielen anderen Sprachen sind die Bedingung, die Verzweigung, die Auswahl, die Schleife und der Sprung.

Sequenz Die einfachste Kontrollanweisung ist keine, die Anweisungen im Programm werden der Reihe nach abgearbeitet. Diese Ablaufform heißt **Sequenz** oder Folge und wird der Vollständigkeit halber erwähnt.

Bedingung Die Ausführung eines Blocks kann von einer **Bedingung** (condition) abhängig gemacht werden. Die Bedingung ist ein Ausdruck, der nur die Werte `true` oder `false` annimmt. Ist die Bedingung `true`, wird der Block abgearbeitet und dann im Programm fortgefahren. Ist die Bedingung `false`, wird der Block übersprungen. Kann die Bedingung niemals `true` werden, hat man toten (unerreichbaren) Code geschrieben. Ist die Bedingung immer `true`, sollte man auf sie verzichten.

In C wird die Bedingung mit dem Schlüsselwort `if` eingeleitet, ohne `then` (im Unterschied zu einem Shellscript). Besteht der Block nur aus einer einzigen Anweisung, kann auf die geschweiften Klammern verzichtet werden:

```
if (Ausdruck) einzelne_Anweisung;  /* oder */
if (Ausdruck) {Block von Anweisungen}
```

Verzweigung (C) Bei einer **Verzweigung** (branch) entscheidet sich der Computer, in Abhängigkeit von einer Bedingung in einem von zwei Programmzweigen weiterzumachen. Im Gegensatz zur Schleife kommt kein Rücksprung vor. Verzweigungen dürfen geschachtelt werden. Dem Computer macht das nichts aus, aber vielleicht verlieren Sie die Übersicht.

Oft, aber nicht notwendigerweise treffen die beiden Zweige im weiteren Verlauf wieder zusammen. Die Syntax sieht folgendermaßen aus:

```
if (Ausdruck) {Block 1}
else {Block 2}
```

Es wird also stets entweder Block 1 oder Block 2 ausgeführt.

Auswahl Stehen am Verzweigungspunkt mehr als zwei Wege offen, so spricht man von einer **Auswahl** (selection). Sie lässt sich grundsätzlich durch eine Schachtelung einfacher Verzweigungen mit `if - else` darstellen, das ist jedoch unübersichtlich.

Zur Konstruktion einer Auswahl braucht man die Schlüsselwörter `switch`, `case`, `break` und `default`. Die Syntax ist die folgende:

```
switch(x) {
        case a:
                Anweisungsfolge 1
                break;
        case b:
        case c:
                Anweisungsfolge 2
                break;
        default:
                Anweisungsfolge 3
}
```

Die Variable `x` (nur vom Typ `int` oder `char`) wird auf Übereinstimmung mit der typgleichen Konstanten `a` geprüft. Falls ja, wird die Anweisungsfolge 1 ausgeführt und infolge der `break`-Anweisung die Auswahl verlassen. Stimmt `x` nicht mit `a` überein oder fehlt nach `case a` das `break`, wird dann `x` auf Übereinstimmung mit `b` oder `c` geprüft. Trifft kein `case` zu, wird die `default`-Anweisungsfolge ausgeführt. Fehlt diese, macht das Programm nach der Auswahl weiter, ohne eine der Anweisungen ausgeführt zu haben. Wenn keine anderen Gründe dagegen sprechen, stellt man den häufigsten Fall an den Anfang.

Die Auswahl ist übersichtlich, einfach zu erweitern und effektiv. Wenn aus einer einfachen Verzweigung eine Auswahl werden könnte, soll man gleich zu dieser greifen. Auswahlen dürfen geschachtelt werden.

Mit etwas Phantasie kann man sich die Bedingung als eine Auswahl mit nur einer Wahlmöglichkeit vorstellen, die Verzweigung als eine Auswahl mit zwei Wahlmöglichkeiten. Insofern lassen sich diese drei Kontrollstrukturen zusammenfassen, wobei die `switch`-Auswahl den allgemeinen Fall darstellt.

Schleifen Einem Computer macht es nichts aus, denselben Vorgang millionenmal zu wiederholen. Das ist seine Stärke. Wiederholungen von Anweisungen kommen daher in fast allen Programmen vor, sie werden **Schleifen** (loop) genannt.

Eine Schleife hat einen **Eingang**, sonst käme man nicht hinein. Die meisten Schleifen haben auch einen **Ausgang**, sonst käme man nicht wieder heraus (außer mit dem Brecheisen in Form der Break-Taste oder Ähnlichem).

Entweder Ein- oder Ausgang sind an eine **Bedingung** geknüpft, die entscheidet, wie oft die Schleife durchlaufen wird. Folgende Konstruktionen sind möglich:

- Eingang: Eintrittsbedingung
- Schleifenrumpf (Anweisungen)
- Ausgang: Rücksprung zum Eingang

Diese Schleife wird nur betreten, falls die Eintrittsbedingung erfüllt ist, unter Umständen also nie. Sie wird deshalb **abweisende Schleife** genannt, auch kopfgesteuerte Schleife. Wenn die Eintrittsbedingung nicht mehr erfüllt ist, macht das Programm nach der Schleife weiter. In C sieht diese Schleife so aus:

```
while (Bedingung) einzelne_Anweisung;   /* oder */
while (Bedingung) {Block von Anweisungen}
}
```

Die zweite Möglichkeit lässt sich grundsätzlich auf die erste zurückführen, wird aber trotzdem verwendet, weil das Programm dadurch einfacher wird:

- Eingang (wird in jedem Fall betreten)
- Schleifenrumpf (Anweisungen)
- Ausgang: Rücksprungbedingung

Diese Schleife wird also mindestens einmal ausgeführt und dann so lange wiederholt, wie die Rücksprungbedingung zutrifft. Sie heißt daher **nichtabweisende Schleife**, auch fußgesteuerte Schleife. Ist die Rücksprungbedingung nicht mehr erfüllt, macht das Programm nach der Schleife weiter. In C:

```
do einzelne_Anweisung while (Bedingung);  /* oder */
do {Block von Anweisungen} while (Bedingung);
```

Eine Variante, die eine `#define`-Zeile erfordert, sieht folgendemaßen aus:

```
#define Please
...
Please do {Block} while (Bedingung);
```

In hartnäckigen Fällen soll diese Schleife der Standard-Schleife überlegen sein.

Rein aus Bequemlichkeit gibt es in C noch eine dritte Schleife mit `for`, die aber stets durch eine `while`-Schleife ersetzt werden kann. Sie sieht so aus:

```
for (Initialisierung; Bedingung; Inkrementierung) {
Block von Anweisungen;
}
```

Vor Eintritt in die Schleife wird der Ausdruck `initialisierung` ausgewertet (also immer), dann wird der Ausdruck `bedingung` geprüft und falls ungleich 0 der Schleifenrumpf betreten. Zuletzt wird der Ausdruck `inkrementierung` ausgewertet und zur Bedingung zurückgesprungen. Die `for`-Schleife in C hat also eine andere Bedeutung als die `for`-Schleife der Shell oder der Programmiersprache PASCAL. Jeder der drei Ausdrücke darf fehlen:

```
for (;;);
```

ist die ewige und zugleich leere Schleife. Die Initialisierung und die Inkrementierung dürfen mehrere, durch den Komma-Operator getrennte Ausdrücke enthalten, die Bedingung muss einen Wert gleich oder ungleich 0 ergeben. Zwei Beispiele:

```
/* Beispiel fuer for-Schleife, 04.03.1993 */

#define  MAX  10
#include <stdio.h>

int main()
{
int i;
for (i = 0; i < MAX; i++)
    printf("Der Schleifenzaehler spricht: %d\n", i);
return 0;
}
```

Quelle 1.28 : C-Programm mit einfacher for-Schleife

Der Schleifenzähler `i` wird mit 0 initialisiert. Für `MAX` ist bereits vom Compiler die Zahl 10 eingesetzt worden; die Eintrittsbedingung `i < 10` ist anfangs erfüllt, der Schleifenrumpf wird ausgeführt. Dann wird der dritte Teil der `for`-Zeile ausgeführt, nämlich der Schleifenzähler `i` um 1 erhöht, und zur Bedingung `i < 10` zurückgesprungen. Das wiederholt sich, bis `i` den Wert 10 erreicht hat. Die Bedingung ist dann nicht mehr erfüllt, die Ausführung des Programms geht nach der Schleife weiter. Nun der syntaktisch einwandfreie Mißbrauch einer `for`-Schleife:

```
/* Testen der for-Schleife, 04.03.1993 */

#define  MAX  10
#include <stdio.h>

int sum(int x);

int main()
{
int i, j = 1;

for (i = - 3, puts("Anfang"); i < j * MAX; i++, i = sum(i))
{
  printf("Der Schleifenzaehler spricht: %d %d\n", i, j);
  j = (i < 0 ? -i : 3);
}
return i;
}

/* Funktion sum(x) */

int sum(int x)
{
if (x < 5) return (x + 1);
else return (x + 2);
}
```

Quelle 1.29 : C-Programm mit zusammengesetzter for-Schleife

Im Initialisierungsteil wird der Schleifenzähler `i` mit -3 belegt und – getrennt durch den Komma-Operator – mittels der Standard-Funktion `puts(3)` ein String ausgegeben. In der Eintrittsbedingung wird gerechnet; wichtig ist nur, dass schließlich ein Wert 0 oder nicht-0 herauskommt. Dann wird gegebenenfalls der Schleifenrumpf ausgeführt, wobei im Rumpf auf die Variablen `i` und `j` des Schleifenkopfes zugegriffen wird. Abschließend wird der Schlei-

fenzähler i inkrementiert und – wieder durch den Komma-Operator getrennt – nochmals mit Hilfe einer Funktion sum(x) verändert. Wenn die Schleife nach einigen Durchläufen verlassen wird, steht der Schleifenzähler i weiterhin zur Verfügung. In dem Kopf der for-Schleife lässt sich allerhand unterbringen, auch Anweisungen, die mit der Schleife nichts zu tun haben. Das wäre schlechter Stil.

Ist die Eintritts- oder Rücksprungbedingung immer erfüllt, bleibt der Computer in der Schleife gefangen, man hat eine ewige Schleife programmiert. Das kann gewollt sein, ist aber oft ein Programmierfehler.

Schleifen mit der Bedingung mitten im Schleifenrumpf sind denkbar und kommen vor, jedoch selten. Mehrere Ausgänge sind erlaubt, verringern aber die Übersicht und sind sparsam zu verwenden. Bei der Behandlung von Ausnahmen (Division durch Null) braucht man sie manchmal. Das Hineinspringen mitten in den Schleifenrumpf ist möglich, gilt aber als schwerer Stilfehler.

Die Anweisung break im Rumpf führt zum sofortigen und endgültigen Verlassen einer Schleife. Die Anweisung continue bricht den augenblicklichen Durchlauf ab und springt zurück vor die Schleife, bei der for-Schleife vor die Initialisierung. Der Systemaufruf exit(2) veranlasst den sofortigen Abbruch des ganzen Programmes, ist also für unheilbare Ausnahmezustände zu gebrauchen (Notschlachtung).

In vielen Schleifen zählt man die Anzahl der Durchläufe (und verzählt sich dabei oft um eins[30]). Die zugehörige Variable ist der **Schleifenzähler**. Auf seine Initialisierung ist zu achten. Der Schleifenzähler steht in und nach der Schleife für Rechnungen zur Verfügung, anders als in FORTRAN.

Schleifen dürfen geschachtelt werden. Mit mehrfach geschachtelten Schleifen geht der Spaß erst richtig los. Der Rumpf der innersten Schleife wird am häufigsten durchlaufen und hat daher auf das Zeitverhalten des Programmes einen großen Einfluss. Dort sollte man nur die allernötigsten Anweisungen hineinschreiben. Auch die Bedingung der innersten Schleife sollte so einfach und knapp wie möglich gefasst sein.

Sprung Es gibt die Anweisung goto, gefolgt von einem Label (Marke). In seltenen Fällen kann ein goto das Programm verbesern, meist ist es vom Übel, weil es erstens sehr gefährlich, zweitens auch nicht nötig ist[31].

Hier ein grauenvolles Beispiel für den Mißbrauch von goto. Das Programm ist syntaktisch richtig und tut auch das, was es soll, nämlich die eingegebene Zahl ausgeben, falls sie durch 5 teilbar ist, andernfalls die nächstgrößere durch 5 teilbare Zahl einschließlich der Zwischenergebnisse. Das Programm enthält eine schwere programmiertechnische Sünde, den Sprung mitten in einen Schleifenrumpf:

```
/* Grauenvolles Beispiel fuer goto, 06.07.1992 */
/* Am besten gar nicht compilieren */

#include <stdio.h>

int main()
{
int x;

printf("Bitte Zahl eingeben: ");
scanf("%d", &x);

if (!(x % 5))
    goto marke;
else {
    while (x % 5) {
        x++;
        marke:
        printf("Ausgabe: %d\n", x);
```

[30]Sogenannter Zaunpfahl-Fehler (fencepost error): Wenn Sie bei einem Zaun von 100 m Länge alle 10 m einen Pfahl setzen, wieviele Pfähle brauchen Sie? 9? 10? 11?

[31]Real programmers aren't afraid to use goto's.

```
    }
}
return 0;
}
```

Quelle 1.30 : C-Programm mit goto, grauenvoll

Nun aber ganz schnell eine stilistisch einwandfreie Fassung des Programms:

```
/* Verbessertes Beispiel, 06.07.1992 */

#include <stdio.h>

int main()
{
int x;

printf("Bitte Zahl eingeben: ");
scanf("%d", &x);

do {
    printf("%d\n", x);
} while (x++ % 5);

return 0;
}
```

Quelle 1.31 : C-Programm, verbessert

Am `goto` hatte sich um 1970 herum ein Glaubenskrieg entzündet. In C-Programmen besteht äußerst selten die Notwendigkeit für diese Anweisung, aber gebräuchliche Anweisungen wie `break`, `continue` und `return` sind bei Licht besehen auch nur `goto`s, die auf bestimmte Fälle beschränkt sind. Immerhin verhindern die Beschränkungen ein hemmungsloses Hinundherhüpfen im Programm.

1.3.8.4 Rückgabewert

Eine Funktion braucht keinen Wert an die aufrufende Einheit zurückzugeben. Sie ist dann vom Typ `void`. Ihre Bedeutung liegt allein in dem, was sie tut, zum Beispiel den Bildschirm putzen. In diesem Fall endet sie ohne `return`-Anweisung (schlechter Stil) oder mit einer `return`-Anweisung ohne Argument. Was sie tut, wird **Nebenwirkung** oder **Seiteneffekt** (side effect) genannt. In FORTRAN wäre das eine Subroutine, in PASCAL eine eigentliche Prozedur.

Gibt man der `return`-Anweisung einen Wert mit, so kann die Funktion von der aufrufenden Einheit wie ein Ausdruck angesehen werden. Der **Rückgabewert** (return value) darf nur ein einfacher Datentyp oder ein Pointer sein. Will man einen String zurückgeben, geht das nur über den Pointer auf den Anfang des Strings. Der zurückzugebende Wert braucht nicht eingeklammert zu werden; bei zusammengesetzten Ausdrücken sollte man der Lesbarkeit halber Klammern setzen:

```
return 0;
return (x + y);
return arrayname;
```

Besteht das Ergebnis aus mehreren Werten, so muss man mit globalen Variablen oder mit Pointern arbeiten. Der Rückgabewert kann immer nur ein einziger Wert sein.

Es kommt vor, dass eine Funktion zwar einen Wert zurückgibt, dieser aber nicht weiter verwendet wird. In diesem Fall warnt `lint(1)`, aber das Programm ist korrekt. Häufig bei `printf(3)` und Verwandtschaft. Den Rückgabewert der Funktion `main()` findet man in der Shell-Variablen `$?` oder `$status`. Er kann in einem Shellscript weiterverarbeitet werden. Hier ein Beispiel für den Gebrauch der `return`-Anweisung:

```
/* Beispiel fuer return-Anweisungen, 21.02.91 */

#define PI 3.14159
#include <stdio.h>

/* Funktionsdeklarationen (Prototypen) */

void text(); int eingabe(); double area(float rad);
char *maxi();

/* Hauptprogramm */

int main()
{
float r; char w1[63], w2[63];

text();

if (!eingabe())
    puts("Eingabe war richtig.");
else
    puts("Eingabe war falsch.");

printf("Radius eingeben: ");
scanf("%f", &r);
printf("Kreisflaeche: %lf\n", area(r));

printf("Bitte zwei Woerter eingeben: ");
scanf("%s %s", w1, w2);
printf("Das laengere Wort ist:  %s\n", maxi(w1, w2));

return 0;
}

/* Funktion ohne Returnwert, Typ void */

void text()
{
puts("\nDiese Funktion gibt nichts zurueck.");
return;
}

/* Funktion mit richtig/falsch-Returnwert, Typ int */

int eingabe()
{
int i;
printf("Bitte die Zahl 37 eingeben: ");
scanf("%d", &i);
if (i == 37) return 0;
else return -1;
}

/* Funktion, die ein Rechenergebnis liefert, Typ double */

double area(float rad)
{
return (rad * rad * PI);
}

/* Funktion, die einen Pointer zurueckgibt, Typ (char *) */

char *maxi(char *w1,char *w2)
{
int i, j;
for (i = 0; w1[i] != '\0'; i++) ;
```

```
for (j = 0; w2[j] != '\0'; j++) ;
return((j > i) ? w2 : w1);
}
```

Quelle 1.32 : C-Programm mit return-Anweisungen

Im Hauptprogramm `main()` haben `return(n);` und `exit(n);` dieselbe Wirkung. In anderen Funktionen führt `return` zur Rückkehr in die nächsthöhere Einheit, `exit` zum Abbruch des gesamten Programmes. In der Syntax unterscheiden sich beide Aufrufe: `return` ist ein Schlüsselwort von C, `exit()` ein Systemaufruf von UNIX, also eine Funktion. Weiterhin sind `exit` und `return` auch eingebaute Shell-Kommandos – siehe `sh(1)` oder `ksh(1)` – die aber nicht in C-Programmen vorkommen können.

1.3.9 Memo Bausteine

- Kommentar ist für den menschlichen Leser bestimmt, der Compiler übergeht ihn oder bekommt ihn gar nicht erst zu Gesicht.
- Namen bezeichnen Funktionen, Konstanten, Variable aller Art, Makros oder Sprungmarken (Labels). Sie sollen mit einem Buchstaben beginnen.
- Operanden haben Namen, Typ, Geltungsbereich, Lebensdauer und spätestens bei ihrer erstmaligen Benutzung einen Wert und belegen damit einen Platz und eine Adresse im Speicher.
- Eine Vereinbarung besteht aus Deklaration und Definition. Die Deklaration weist einem Operanden Name und Typ zu und legt seinen Geltungsbereich und seine Lebensdauer fest. Mit der Definition erhält ein Operand einen Wert. Beides kann in einer Anweisung vereinigt werden.
- Der Typ entscheidet über Wertebereich, Operationen und Speicherbedarf. Der Typ eines Operanden ist in C konstant, er ändert sich während der Programmausführung nicht.
- Es gibt einfache und zusammengesetzte Typen sowie Pointer.
- Einfache Typen sind Ganzzahlen, Gleitkommazahlen und Zeichen. Der Aufzählungstyp ist letzten Endes ganzzahlig. Daneben gibt es für bestimmte Fälle den leeren Typ.
- Zusammengesetzte Typen sind Arrays und Strukturen.
- Ein Array ist eine geordnete Menge von Operanden gleichen Typs.
- Eine Struktur ist eine ungeordnete Menge von Operanden beliebigen Typs.
- Der Pointer ist ein Typ, dessen Wertebereich Adressen sind. Ein Pointer zeigt immer auf einen Typ, unter Umständen auf einen weiteren Pointer.
- Ein Ausdruck besteht aus Operanden und Operationen. Er hat einen Wert und kann überall dort stehen, wo ein Wert verlangt wird.
- Die einfachste Operation ist die Zuweisung, nicht zu verwechseln mit einer mathematischen Gleichung. Sie hat eine linke und eine rechte Seite. Rechts steht immer ein Wert, also gegebenenfalls ein Ausdruck. Links steht eine Variable oder ein Pointer, aber niemals ein Ausdruck.
- Weiterhin gibt es arithmetische, logische, vergleichende Operationen sowie O. auf Bits, Strukturen oder Pointer. Dann haben wir noch den Cast-, Komma- und Sizeof-Operator.
- Anweisungen enden immer mit einem Semikolon.
- Aus einer Zuweisung wird durch Anfügen eines Semikolons eine Anweisung.
- Die Kontrollanweisungen Bedingung (if), Verzweigung (if-else), Auswahl (switch), Schleife (while, do-while, for) und Sprung (break, continue, return, goto) steuern den Programmablauf.

1.3.10 Übung Bausteine

Überlegen Sie, welche Operanden mit welchen Typen Sie für das Beispiel der Weganalyse (oder des Vokabeltrainers) aus dem vorigen Abschnitt brauchen. Eine gute Datenstruktur ist schon fast das halbe Programm. Um ein richtiges Programm schreiben zu können, fehlt uns noch der nächste Abschnitt.

1.4 Funktionen

1.4.1 Aufbau und Deklaration

In C ist eine **Funktion** eine abgeschlossene Programmeinheit, die mit der Außenwelt über einen Eingang und wenige Ausgänge – gegebenenfalls noch Notausgänge – verbunden ist. Hauptprogramm, Unterprogramme, Subroutinen, Prozeduren usw. sind in C allesamt Funktionen. Eine Funktion ist die kleinste kompilierbare Einheit (nicht: ausführbare Einheit, das ist ein Programm), nämlich dann, wenn sie zugleich allein in einer Datei steht. Mit weniger als einer Funktion kann der Compiler nichts anfangen.

Da die **Definitionen von Funktionen** nicht geschachtelt werden dürfen (wohl aber ihre Aufrufe), gelten Funktionen grundsätzlich global. In einem C-Programm stehen alle Funktionen einschließlich `main()` auf gleicher Stufe. Das ist ein wesentlicher Unterschied zu PASCAL, wo Funktionen innerhalb von Unterprogrammen definiert werden dürfen. In C gibt es zu einer Funktion keine übergeordnete Funktion, deren Variable in der untergeordneten Funktion gültig sind.

Eine Funktion übernimmt von der aufrufenden Anweisung einen festgelegten Satz von Argumenten oder Parametern, tut etwas und gibt keinen oder genau einen Wert an die aufrufende Anweisung zurück.

Vor dem ersten Aufruf einer Funktion muß ihr Typ (d. h. der Typ ihres Rückgabewertes) bekannt sein. Andernfalls nimmt der Compiler den Standardtyp `int` an. Entsprechend dem ANSI-Vorschlag bürgert es sich zunehmend ein, Funktionen durch ausführliche **Prototypen** vor Beginn der Funktion `main()` zu deklarieren:

```
/* Beispiel fuer Funktionsprototyp */

float multipl(float x, float y);     /* Prototyp */

/* es reicht auch: float multipl(float, float); */
/* frueher nach K+R: float multipl(); */

int main()
{
float a, b, z;
.
.
z = multipl(a, b);                /* Funktionsaufruf */
.
.
}

float multipl(float x, float y)   /* F'definition */
{
return (x * y);
}
```

Quelle 1.33 : C-Programm mit Funktionsprototyp

Durch die Angabe der Typen der Funktion und ihrer Argumente zu Beginn des Programms herrscht sofort Klarheit. Die Namen der Parameter sind unerheblich; Anzahl, Typ und Reihenfolge sind wesentlich. Noch nicht alle Compiler unterstützen die Angabe der Argumenttypen. Auch den Standardtyp `int` sollte man deklarieren, um zu verdeutlichen, daß man ihn nicht vergessen hat. Änderungen werden erleichtert.

1.4.2 Pointer auf Funktionen

Der **Name** einer Funktion ohne die beiden runden Klammern ist der Pointer auf ihren Eingang (entry point). Damit kann ein Funktionsname überall verwendet werden, wo Pointer zulässig sind. Insbesondere kann er als Argument einer weiteren Funktion dienen. In funktionalen Programmiersprachen ist die Möglichkeit, Funktionen als Argumente höherer Funktionen zu verwenden, noch weiter entwickelt. Arrays von Funktionen sind nicht zulässig, wohl aber Arrays von Pointern auf Funktionen, siehe Programm 1.90 auf Seite 155.

Makros (`#define ...`) sind keine Funktionen, infolgedessen gibt es auch keine Pointer auf Makros. Zu Makros siehe Abschnitt 1.9 *Präprozessor* auf Seite 129.

1.4.3 Parameterübergabe

Um einer Funktion die Argumente oder Parameter zu übermitteln, gibt es mehrere Wege. Grundsätzlich müssen in der Funktion die entsprechenden Variablen als Platzhalter oder **formale Parameter** vorkommen und deklariert sein. Im Aufruf der Funktion kommt der gleiche Satz von Variablen – gegebenenfalls unter anderem Namen – mit jeweils aktuellen Werten vor; sie werden als **aktuelle Parameter** oder Argumente bezeichnet. Die Schnittstelle von Programm und Funktion muß zusammenpassen wie Stecker und Kupplung einer elektrischen Verbindung, d. h. die Liste der aktuellen Parameter muß mit der Liste der formalen Parameter nach Anzahl, Reihenfolge und Typ der Parameter übereinstimmen.

Bei der **Wertübergabe** (call by value) wird der Funktion eine Kopie der aktuellen Parameter des aufrufenden Programmes übergeben. Daraus folgt, daß die Funktion die aktuellen Parameter des aufrufenden Programmes nicht verändern kann.

Bei der **Adressübergabe** (call by reference) werden der Funktion die Speicheradressen der aktuellen Parameter des aufrufenden Programmes übergeben. Die Funktion kann daher die Werte der Parameter mit Wirkung für das aufrufende Programm verändern. Sie arbeitet mit den Originalen der Parameter. Das kann erwünscht sein oder auch nicht. Bei beiden Mechanismen werden die Parameter vollständig ausgerechnet, ehe die Funktion betreten wird.

Wie die Parameterübergabe in C, FORTRAN und PASCAL aussieht, entnimmt man am besten den Beispielen. Die Parameter sind vom Typ integer, um die Beispiele einfach zu halten. Ferner ist noch ein Shellscript angegeben, das eine C-Funktion aufruft, die in diesem Fall ein selbständiges Programm (Funktion `main()`) sein muß.

Der von einer Funktion zurückgegebene Wert (**Rückgabewert**) kann nur ein einfacher Typ oder ein Pointer sein. Zusammengesetzte Typen wie Arrays, Strings oder Strukturen können nur durch Pointer zurückgegeben werden. Es ist zulässig, keinen Wert zurückzugeben. Dann ist die Funktion vom Typ void und macht sich allein durch ihre Nebeneffekte bemerkbar.

Für die Systemaufrufe von UNIX und die Standardfunktionen von C ist im Referenz-Handbuch in den Sektionen (2) und (3) angegeben, von welchem Typ die Argumente und der Funktionswert sind. Da diese Funktionen allesamt C-Funktionen sind, lassen sie sich ohne Probleme in C-Programme einbinden. Bei anderen Sprachen ist es denkbar, daß kein einem C-Typ entsprechender Variablentyp verfügbar ist. Auch bei Strings gibt es wegen der unterschiedlichen Speicherung in den einzelnen Sprachen Reibereien. Falls die Übergabemechanismen unverträglich sind, muß man die C-Funktion in eine Funktion oder Prozedur der anderen Sprache so verpacken, daß das aufrufende Programm eine einheitliche Programmiersprache sieht. Das Vorgehen dabei kann maschinenbezogen sein, was man eigentlich vermeiden will.

In den folgenden Programmbeispielen wird die Summe aus zwei Summanden berechnet, zuerst im Hauptprogramm direkt und dann durch zwei Funktionen, die ihre Argumente – die Summanden – by value beziehungsweise by reference übernehmen. Die Funktionen verändern ihre Summanden, was im ersten Fall keine Auswirkung im Hauptprogramm hat. Hauptprogramme und Funktionen sind in C, FORTRAN und PASCAL geschrieben, was neun Kombinationen ergibt. Wir betreten damit zugleich das an Fallgruben reiche Gebiet der Mischung von Programmiersprachen (mixed language programming). Zunächst die beiden Funktionen im geliebten C:

```
/* C-Funktion (Summe) call by value */
/* Compileraufruf cc -c csv.c, liefert csv.o */

int csv(int x,int y)
{
int z;
puts("Funktion mit Parameteruebernahme by value:");
printf("C-Fkt. hat uebernommen:    %d   %d\n", x, y);
z = x + y;
printf("C-Fkt. gibt folgende Summe zurueck: %d\n", z);
/* Aenderung der Summanden */
x = 77; y = 99;
return(z);
}
```

Quelle 1.34 : C-Funktion, die Parameter by value übernimmt

```
/* C-Funktion (Summe) call by reference */
/* Compileraufruf cc -c csr.c, liefert csr.o */

int csr(int *px,int *py)
{
int z;
puts("Funktion mit Parameteruebenahme by reference:");
printf("C-Fkt. hat uebernommen: %d   %d\n", *px, *py);
z = *px + *py;
printf("C-Fkt. gibt folgende Summe zurueck:  %d\n", z);
/* Aenderung der Summanden */
*px = 66; *py = 88;
return(z);
}
```

Quelle 1.35 : C-Funktion, die Parameter by reference übernimmt

Im bewährten FORTRAN 77 haben wir leider keinen Weg gefunden, der Funktion beizubringen, ihre Parameter by value zu übernehmen (in FORTRAN 90 ist es möglich). Es bleibt daher bei nur einer Funktion, die – wie in FORTRAN üblich – ihre Parameter by reference übernimmt:

```
C     Fortran-Funktion (Summe) call by reference
C     Compileraufruf f77 -c fsr.f

      integer function fsr(x, y)
      integer x, y, z

      write (6,'("F-Fkt. mit Uebernahme by reference:")')
      write (6,'("F-Fkt. hat uebernommen: ",2I6)') x, y
      z = x + y
      write (6,'("F-Fkt. gibt zurueck: ",I8)') z
C     Aenderung der Summanden
      x = 66
      y = 88
      fsr = z

      end
```

Quelle 1.36 : FORTRAN-Funktion, die Parameter by reference übernimmt

PASCAL-Funktionen kennen wieder beide Möglichkeiten, aber wir werden auf eine andere Schwierigkeit stoßen. Vorläufig sind wir jedoch hoffnungsvoll:

```
{Pascal-Funktion (Summe) call by value}
{Compileraufruf pc -c psv.p}
```

```
module b;
import StdOutput;
export
    function psv(x, y: integer): integer;
implement
    function psv;
    var z: integer;
    begin
        writeln('Funktion mit P'uebernahme by value:');
        writeln('P-Fkt. hat uebernommen: ', x, y);
        z := x + y;
        writeln('P-Fkt. gibt folgenden Wert zurueck: ', z);
        { Aenderung der Summanden }
        x := 77; y := 99;
        psv := z;
    end;
end.
```

Quelle 1.37 : PASCAL-Funktion, die Parameter by value übernimmt

```
{Pascal-Funktion (Summe) call by reference}
{Compileraufruf pc -c psr.p}

module a;
import StdOutput;
export
    function psr(var x, y: integer): integer;
implement
    function psr;
    var z: integer;
    begin
        writeln('Funktion mit P'uebernahme by reference:');
        writeln('P-Fkt. hat uebernommen: ', x, y );
        z := x + y;
        writeln('P-Fkt. gibt folgenden Wert zurueck: ', z);
        { Aenderung der Summanden }
        x := 66; y := 88;
        psr := z;
    end;
end.
```

Quelle 1.38 : PASCAL-Funktion, die Parameter by reference übernimmt

Die Funktionen werden für sich mit der Option `-c` ihres jeweiligen Compilers kompiliert, wodurch Objektfiles mit der Kennung `.o` entstehen, die beim Kompilieren der Hauptprogramme aufgeführt werden. Nun zu den Hauptprogrammen, zuerst wieder in C:

```
/* C-Programm csummec, das C-Funktionen aufruft */
/* Compileraufruf cc -o csummec csummec.c csr.o csv.o */

#include <stdio.h>

extern int csv(int x,int y),
           csr(int *px,int *py);

int main()
{
int a, b;
puts("Bitte die beiden Summanden eingeben!");
scanf("%d %d", &a, &b);
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe (direkt) ist:   %d\n", (a + b));
printf("Die Summe ist: %d\n", csv(a, b));
```

```
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe ist:  %d\n", csr(&a, &b));
printf("Die Summanden sind:  %d   %d\n", a, b);
return 0;
}
```

Quelle 1.39 : C-Programm, das Parameter by value und by reference an C-Funktionen übergibt

Nun das C-Hauptprogramm, das eine FORTRAN-Funktion aufruft, ein in der Numerik häufiger Fall:

```
/* C-Programm csummef, das eine FORTRAN-Funktion aufruft */
/* Compileraufruf cc -o csummef csummef.c fsr.o -lcl */

#include <stdio.h>

extern int fsr(int *x,int *y);

int main()
{
int a, b;
scanf("%d %d", &a, &b);
printf("Die Summanden sind:  %d   %d\n", a, b);
printf("Die Summe (direkt) ist:   %d\n", (a + b));
printf("Die Summe ist:  %d\n", fsr(&a, &b));
printf("Die Summanden sind:  %d   %d\n", a, b);
return 0;
}
```

Quelle 1.40 : C-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

Die Linker-Option `-lcl` ist erforderlich, wenn FORTRAN- oder PASCAL-Module in C-Programme eingebunden werden. Sie bewirkt die Hinzunahme der FORTRAN- und PASCAL-Laufzeitbibliothek `/usr/lib/libcl.a`, ohne die Bezüge (Referenzen) auf FORTRAN- oder PASCAL-Routinen unter Umständen offen bleiben. Anders gesagt, in den FORTRAN- oder PASCAL-Funktionen kommen Namen vor – zum Beispiel `write` – deren Definition in besagter Laufzeitbibliothek zu finden ist. C und PASCAL sind sich im großen ganzen ähnlich, es gibt aber Unterschiede hinsichtlich des Geltungsbereiches von Variablen, die hier nicht deutlich werden:

```
/* C-Programm csummep, das PASCAL-Funktionen aufruft. */
/* Compiler: cc -o csummep csummep.c psv.o psr.o -lcl */

#include <stdio.h>

extern int psv(int x,int y),psr(int *x,int *y)

int main()
{
int a, b;
puts("Bitte die beiden Summanden eingeben!");
scanf("%d %d", &a, &b);
printf("Die Summanden sind: %d   %d\n", a, b);
printf("Die Summe (direkt) ist: %d\n", (a + b));
printf("Die Summe ist: %d\n", psv(a, b));
printf("Die Summanden sind: %d   %d\n", a, b);
printf("Die Summe ist: %d\n", psr(&a, &b));
printf("Die Summanden sind: %d   %d\n", a, b);
return 0;
}
```

Quelle 1.41 : C-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Hiernach sollte klar sein, warum die C-Standardfunktion `printf(3)` mit Variablen als Argument arbeitet, während die ähnliche C-Standardfunktion `scanf(3)` Pointer als Argument verlangt. `printf(3)` gibt Werte aus, ohne sie zu ändern. Es ist für das Ergebnis belanglos, ob die Funktion Adressen (Pointer) oder Kopien der Variablen verwendet (die Syntax legt das allerdings fest). Hingegen soll `scanf(3)` Werte mit Wirkung für die aufrufende Funktion einlesen. Falls es sich nur um einen Wert handelte, könnte das noch über den Returnwert bewerkstelligt werden, aber `scanf(3)` soll mehrere Werte – dazu noch verschiedenen Typs – verarbeiten. Das geht nur über von `scanf(3)` und der aufrufenden Funktion gemeinsam verwendete Pointer.

Nun die drei FORTRAN-Hauptprogramme mit Aufruf der Funktionen in C, FORTRAN und PASCAL:

```
C      FORTRAN-Programm, das C-Funktionen aufruft
C      Compileraufruf f77 -o fsummec fummec.f csv.o csr.o

       program fsummec

$ALIAS csv (%val, %val)

       integer a, b , s, csr, csv

       write (6, 100)
       read (5, *) a, b
       write (6, 102) a, b
       s = a + b
       write (6, 103) s
C      call by value
       s = csv(a, b)
       write (6, 104) s
       write (6, 102) a, b
C      call by reference
       s = csr(a, b)
       write (6, 105) s
       write (6, 102) a, b

  100 format ('Bitte die beiden Summanden eingeben!')
  102 format ('Die Summanden sind: ', 2I6)
  103 format ('Die Summe (direkt) ist: ', I8)
  104 format ('Die Summe ist: ', I8)
  105 format ('Die Summe ist: ', I8)

       end
```

Quelle 1.42 : FORTRAN-Programm, das Parameter by value und by reference an C-Funktionen übergibt

```
C      FORTRAN-Programm, das F77-Funktion aufruft
C      Compileraufruf f77 -o fsummef fsummef.f fsr.o

       program fsummef

       integer a, b , s, fsr

       write (6, 100)
       read (5, *) a, b
       write (6, 102) a, b
       s = a + b
       write (6, 103) s
C      call by value nicht moeglich
```

```
C     call by reference (default)
      s = fsr(a, b)
      write (6, 105) s
      write (6, 102) a, b

  100 format ('Bitte die beiden Summanden eingeben!')
  102 format ('Die Summanden sind: ', 2I6)
  103 format ('Die Summe (direkt) ist: ', I8)
  105 format ('Die Summe ist: ', I8)

      end
```

Quelle 1.43 : FORTRAN-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

```
C     FORTRAN-Programm, das PASCAL-Funktionen aufruft
C     Compileraufruf f77 -o fsummep fsummep.f psv.o psr.o

      program fsummep

$ALIAS psv (%val, %val)

      integer a, b, s, psv, psr
      external psv, psr

      write (6, 100)
      read (5, *) a, b
      write (6, 102) a, b
      s = a + b
      write (6, 103) s
C     call by value
      s = psv(a, b)
      write (6, 104) s
      write (6, 102) a, b
C     call by reference
      s = psr(a, b)
      write (6, 105) s
      write (6, 102) a, b

  100 format ('Bitte die beiden Summanden eingeben!')
  102 format ('Die Summanden sind: ', 2I6)
  103 format ('Die Summe (direkt) ist: ', I8)
  104 format ('Die Summe ist: ', I8)
  105 format ('Die Summe ist: ', I8)

      end
```

Quelle 1.44 : FORTRAN-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Die FORTRAN-Compiler-Anweisung `$ALIAS` veranlaßt den Compiler, der jeweiligen Funktion die Parameter entgegen seiner Gewohnheit by value zu übergeben. Zum guten Schluß die PASCAL-Hauptprogramme:

```
{PASCAL-Programm, das C-Funktionen aufruft}
{Compiler: pc -o psummec psummec.p csv.o csr.o}

program psummec (input, output);

var a, b, s: integer;

function csv(x, y: integer): integer;      {call by value}
         external C;
```

```
function csr(var x, y: integer): integer;  {call by ref.}
         external C;

begin
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
write('Die Summe (direkt) ist: '); writeln(s);
s := csv(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
s := csr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 1.45 : PASCAL-Programm, das Parameter by value und by reference an C-Funktionen übergibt

```
{PASCAL-Programm, das FORTRAN-Funktion aufruft}
{Compiler: pc -o psummef psummef.p fsr.o}

program psummef (input, output);

var a, b, s: integer;

function fsr(var x, y: integer): integer;  {call by ref.}
         external ftn77;

begin
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
write('Die Summe (direkt) ist: '); writeln(s);
s := fsr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 1.46 : PASCAL-Programm, das Parameter by reference an eine FORTRAN-Funktion übergibt

```
{PASCAL-Programm, das PASCAL-Funktionen aufruft}
{Compileraufruf pc -o psummep psummep.p psv.o psr.o}

program psummep (input, output);

var a, b, s: integer;

function psv(x, y: integer): integer;      {call by value}
         external;

function psr(var x, y: integer): integer;  {call by ref.}
         external;

begin
writeln('Bitte die beiden Summanden eingeben!');
readln(a); readln(b);
write('Die Summanden sind: '); write(a); writeln(b);
s := a + b;
```

```
write('Die Summe (direkt) ist: '); writeln(s);
s := psv(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
s := psr(a, b);
write('Die Summe ist: '); writeln(s);
write('Die Summanden sind: '); write(a); writeln(b);
end.
```

Quelle 1.47 : PASCAL-Programm, das Parameter by value und by reference an PASCAL-Funktionen übergibt

Sollten Sie die Beispiele nachvollzogen haben, müßte Ihr Linker in zwei Fällen mit einer Fehlermeldung `unsatisfied symbol: output (data)` die Arbeit verweigert haben. Die PASCAL-Funktionen `psv()` und `psr()` geben etwas auf das Terminal aus. Bei getrennt kompilierten Modulen erfordert dies die Zeile:

```
import StdOutput;
```

Das importierte, vorgefertigte PASCAL-Modul `StdOutput` macht von einem Textfile `output` Gebrauch, das letzten Endes der Bildschirm ist. Im PASCAL-Programm öffnet die Zeile

```
program psummep (input, output);
```

dieses Textfile. In C-Programmen wird die Datei mit dem Dateipointer `stdout` ebenso wie in FORTRAN-Programmen die Unit 6 automatisch geöffnet. Hinter dem Dateipointer bzw. der Unit steckt der Bildschirm. Leider sehen wir – in Übereinstimmung mit unseren Handbüchern – keinen Weg, die PASCAL-Datei `output` mit `stdout` von C oder der Unit 6 von FORTRAN zu verbinden. Wollen wir PASCAL-Funktionen in ein C- oder FORTRAN-Programm einbinden, müssen die Funktionen auf Terminalausgabe verzichten (eine Ausgabe in eine Datei wäre möglich):

```
{Pascal-Funktion (Summe) call by value, ohne Output}
{Compileraufruf pc -c xpsv.p}

module b;
export
    function psv(x, y: integer): integer;
implement
    function psv;
    var z: integer;
    begin
        z := x + y;
        { Aenderung der Summanden }
        x := 77; y := 99;
        psv := z;
    end;
end.
```

Quelle 1.48 : PASCAL-Funktion, die Parameter by value übernimmt, ohne Ausgabe

```
{Pascal-Funktion (Summe) call by reference}
{ohne Output}
{Compileraufruf pc -c xpsr.p}

module a;
export
    function psr(var x, y: integer): integer;
implement
    function psr;
    var z: integer;
    begin
        z := x + y;
```

```
        { Aenderung der Summanden }
        x := 66; y := 88;
        psr := z;
    end;
end.
```

Quelle 1.49 : PASCAL-Funktion, die Parameter by reference übernimmt, ohne Ausgabe

Damit geht es. Der Compilerbauer weiß, wie die einzelnen Programmiersprachen ihre Ausgabe bewerkstelligen und kann Übergänge in Form von Compiler-Anweisungen oder Zwischenfunktionen einrichten. So macht es Microsoft bei seinem großen C-Compiler. Aber wenn nichts vorgesehen ist, muß der gewöhnliche Programmierer solche Unverträglichkeiten hinnehmen.

Auch Shellscripts können Funktionen aufrufen. Diese müssen selbständige Programme wie externe Kommandos sein, der Mechanismus sieht etwas anders aus. Hier das Shellscript:

```
# Shellscript, das eine C-Funktion aufruft. 28.01.1988
# Filename shsumme

print Bitte die beiden Summanden eingeben!
read a; read b
print Die Summanden sind $a $b
print Die Shell-Summe ist `expr $a + $b`
print Die Funktions-Summe ist `cssh $a $b`
print Die Summanden sind $a $b
exit
```

Quelle 1.50 : Shellscript mit Parameterübergabe

Die zugehörige C-Funktion ist ein Hauptprogramm:

```
/* C-Programm zum Aufruf durch Shellskript, 29.01.1988 */
/* Compileraufruf: cc -o cssh cssh.c */

int main(int argc, char *argv[])

{
int x, y;
sscanf(argv[1], "%d", &x);
sscanf(argv[2], "%d", &y);
printf("%d", (x + y));
return 0;
}
```

Quelle 1.51 : C-Programm, das Parameter von einem Shellscript übernimmt

Ferner können Shellscripts **Shellfunktionen** aufrufen, siehe das Shellscript **??** *Türme von Hanoi* auf Seite **??**.

Entschuldigen Sie bitte, daß dieser Abschnitt etwas breit geraten ist. Die Parameterübergabe muß sitzen, wenn man mehr als Trivialprogramme schreibt, und man ist nicht immer in der glücklichen Lage, rein in C programmieren zu können. Verwendet man vorgegebene Bibliotheken, so sind diese manchmal in einer anderen Programmiersprache verfaßt. Dann hat man sich mit einer fremden Syntax und den kleinen, aber bedeutsamen Unverträglichkeiten herumzuschlagen.

1.4.4 Kommandozeilenargumente, main()

Auch das Hauptprogramm `main()` ist eine Funktion, die Parameter oder Argumente übernehmen kann, und zwar aus der **Kommandozeile** beim Aufruf des Programms. Sie kennen das von vielen UNIX-Kommandos, die nichts anderes als C-Programme sind.

Der Mechanismus ist stets derselbe. Die Argumente, getrennt durch Spaces oder Ähnliches, werden in ein Array of Strings mit dem Namen `argv` (**Argumentvektor**) gestellt. Gleichzeitig zählt ein **Argumentzähler** `argc` die Anzahl der übergebenen Argumente, wobei der Funktionsname selbst das erste Argument (Index 0) ist. Bei einem Programmaufruf ohne Argumente steht also der Programmname in `argv[0]`, der Argumentzähler `argc` hat den Wert 1. Das erste nichtbelegte Element des Argumentvektors enthält einen leeren String. Die Umwandlung der Argumente vom Typ String in den gewünschten Typ besorgt die Funktion `sscanf(3)`.

Der Anfang eines Hauptprogrammes mit Kommandozeilenargumenten sieht folgendermaßen aus:

```
int main(int argc, char *argv[])

{
char a; int x;

if (argc < 3) {
        puts("Zuwenige Parameter");
        exit(-1);
}
sscanf(argv[1], "%c", &a);
sscanf(argv[2], "%d", &x);
....
```

Quelle 1.52 : C-Programm, das Argumente aus der Kommandozeile übernimmt

Das erste Kommandozeilenargument (nach dem Kommando selbst) wird als Zeichen verarbeitet, das zweite als ganze Zahl. Etwaige weitere Argumente fallen unter den Tisch.

Die Funktion `main()` ist immer vom Typ `extern int`. Da dies der Defaulttyp für Funktionen ist, könnte die Typdeklaration weggelassen werden. Sie kann Argumente übernehmen, braucht es aber nicht. Infolgedessen sind folgende Deklarationen gültig:

```
main()
int main()
extern int main()
main(void)
int main(void)
extern int main(void)
main(int argc, char *argv[])
int main(int argc, char *argv[])
extern int main(int argc, char *argv[])
main(int argc, char **argv)
int main(int argc, char **argv)
extern int main(int argc, char **argv)
```

und alle anderen falsch. Die ersten sechs sind in ihrer Bedeutung gleich, die weiteren gelten bei Argumenten in der Kommandozeile. Die Norm ISO/IEC 9899:1999 sieht für C-Programme, die unter einem Betriebssystem laufen (hosted environment), nur die beiden folgenden Formen vor:

```
int main(void)
int main(int argc, char *argv[])
```

und das reicht auch. Unter POSIX-konformen Betriebssystemen kann ein drittes Argument hinzukommen, das die Umgebung (environment pointer) enthält:

```
int main(int argc, char *argv[], char *envp[])
```

Den Rückgabewert von `main()` sollte man nicht dem Zufall überlassen, sondern mit einer `return`-Anweisung ausdrücklich festlegen (0 bei Erfolg). Er wird von der Shell übernommen.

1.4.5 Funktionen mit wechselnder Argumentanzahl

Mit `main()` haben wir eine Funktion kennengelernt, die eine wechselnde Anzahl von Argumenten übernimmt. Auch für andere Funktionen als `main()` gibt es einen Mechanismus zu diesem Zweck, schauen Sie bitte unter `varargs(5)` nach. Der Mechanismus ist nicht übermäßig intelligent, sondern an einige Voraussetzungen gebunden:

- Es muß mindestens ein Argument vorhanden sein,
- der Typ des ersten Arguments muß bekannt sein,
- es muß ein Kriterium für das Ende der Argumentliste bekannt sein.

Die erforderlichen Makros stehen in den include-Dateien `<varargs.h>` für UNIX System V oder `<stdarg.h>` für ANSI-C. Wir erklären die Vorgehensweise an einem Beispiel, das der Funktion `printf(3)` nachempfunden ist (es ist damit nicht gesagt, daß `printf(3)` tatsächlich so aussieht):

```
/* Funktion printi(), Ersatz fuer printf(), nur fuer
   dezimale Ganzzahlen, Zeichen und Strings. Siehe
   Referenz-Handbuch unter varargs(5), 22.02.91 */
/* Returnwert 0 = ok, -1 = Fehler, sonst wie printf() */
/* Compileraufruf cc -c printi.c */

#include <stdio.h>
#include <varargs.h>

int fputc();
void int_print();

/* Funktion printi(), variable Anzahl von Argumenten */

int printi(va_alist)
va_dcl
{
    va_list pvar;
    unsigned long arg;
    int field, sig;
    char *format, *string;
    long ivar;

/* Uebernahme und Auswertung des Formatstrings */

    va_start(pvar);
    format = va_arg(pvar, char *);

    while (1) {

/* Ausgabe von Literalen */

    while ((*format != '%') && (*format != '\0'))
        fputc(*format++, stdout);

/* Ende des Formatstrings */

    if (*format == '\0') {
        va_end(pvar);
        return 0;
    }

/* Prozentzeichen, Platzhalter */

    format++;
    field = 0;

/* Auswertung Laengenangabe */
```

```
    while (*format >= '0' && *format <='9') {
        field = field * 10 + *format - '0';
        format++;
    }

/* Auswertung Typangabe und Ausgabe des Arguments */

    switch(*format) {
      case 'd':
        sig = ((ivar  = (long)va_arg(pvar, int)) < 0 ? 1 : 0);
        arg = (unsigned long)(ivar < 0 ? -ivar : ivar);
        int_print(arg, sig, field);
        break;
      case 'u':
        arg = (unsigned long)va_arg(pvar, unsigned);
        int_print(arg, 0, field);
        break;
      case 'l':
        switch(*(format + 1)) {
            case 'd':
                sig = ((ivar = va_arg(pvar, long)) < 0 ? 1 : 0);
                arg = (unsigned long)(ivar < 0 ? -ivar : ivar);
                int_print(arg, sig, field);
                break;
            case 'u':
                arg = va_arg(pvar, unsigned long);
                int_print(arg, 0, field);
                break;
            default:
                va_end(pvar);
                return -1;      /* unbekannter Typ */
        }
        format++;
        break;
      case '%':
        fputc(*format, stdout);
        break;
      case 'c':
        fputc(va_arg(pvar, char), stdout);
        break;
      case 's':
        string = va_arg(pvar, char *);
        while ((fputc(*(string++), stdout)) != '\0') ;
        break;
      default:
        va_end(pvar);
        return -1;              /* unbekannter Typ */
    }
    format++;
}
}

/* Funktion zur Ausgabe der dezimalen Ganzzahl */

void int_print(unsigned long number,int signum,int field)

{
    int i;
    char table[21];
    long radix = 10;

    for (i = 0; i < 21; i++)
        *(table + i) = ' ';

/* Umwandlung Zahl nach ASCII-Zeichen */
```

```
    for (i = 0; i < 20; i++) {
        *(table + i) = *("0123456789" + (number % radix));
        number /= radix;
        if (number == 0) break;
    }

/* Vorzeichen */

    if (signum)
        *(table + ++i) = '-';

/* Ausgabe */

    if ((field != 0) && (field < 20))
        i = field -1;

    while (i >= 0)
    fputc(*(table + i-), stdout);
}

/* Ende */
```

Quelle 1.53 : C-Funktion mit wechselnder Anzahl von Argumenten

Nach der include-Datei `varargs.h` folgt in gewohnter Weise die Funktion, hier `printi()`. Ihre Argumentenliste heißt `va_alist` und ist vom Typ `va_dcl`, ohne Semikolon! Innerhalb der Funktion brauchen wir einen Pointer `pvar` auf die Argumente, dieser ist vom Typ `va_list`, nicht zu verwechseln mit der Argumentenliste `va_alist`. Die weiteren Variablen sind unverbindlich.

Zu Beginn der Arbeit muß das Makro `va_start(pvar)` aufgerufen werden. Es initialisiert den Pointer `pvar` mit dem Anfang der Argumentenliste. Am Ende der Arbeit muß entsprechend mit dem Makro `va_end(pvar)` aufgeräumt werden.

Das Makro `va_arg(pvar, type)` gibt das Argument zurück, auf das der Pointer `pvar` zeigt, und zwar in der Form des angegebenen Typs, den man also kennen muß. Gleichzeitig wird der Pointer `pvar` eins weiter geschoben. Die Zeile

```
format = va_arg(pvar, char *);
```

weist dem Pointer auf char `format` die Adresse des Formatstrings in der Argumentenliste von `printi()` zu. Damit ist der Formatstring wie jeder andere String zugänglich. Zugleich wird der Pointer `pvar` auf das nächste Argument gestellt, üblicherweise eine Konstante oder Variable. Aus der Auswertung des Formatstrings ergeben sich Anzahl und Typen der weiteren Argumente.

Damit wird auch klar, was geschieht, wenn die Platzhalter (%d, %6u usw.) im **Formatstring** nicht mit der Argumentenliste übereinstimmen. Gibt es mehr Argumente als Platzhalter, werden sie nicht beachtet. Gibt es mehr Platzhalter als Argumente, wird irgendein undefinierter Speicherinhalt gelesen, unter Umständen auch der dem Programm zugewiesene Speicherbereich verlassen. Stimmen Platzhalter und Argumente im Typ nicht überein, wird der Pointer `pvar` falsch inkrementiert, und die Typumwandlung geht vermutlich auch daneben.

Es gibt eine Fallgrube bei der Typangabe. Je nach Compiler werden die Typen `char` und `short` intern als `int` und `float` als `double` verarbeitet. In solchen Fällen muß dem Makro `va_arg(pvar, type)` der interne Typ mitgeteilt werden. Nachlesen oder ausprobieren, am besten beides.

1.4.6 Iterativer Aufruf einer Funktion

Unter einer **Iteration** versteht man die Wiederholung bestimmter Programmschritte, wobei das Ergebnis eines Schrittes als Eingabe für die nächste Wiederholung dient. Viele mathematische Näherungsverfahren machen von Iterationen Gebrauch. Programmtechnisch führen Iterationen auf Schleifen. Entsprechend muß eine Bedingung angegeben werden, die die

Iteration beendet. Da auch bei einem richtigen Programm eine Iteration manchmal aus mathematischen Gründen nie zu einem Ende kommt, ist es zweckmäßig, einen Test für solche Fälle einzubauen wie in folgendem Beispiel:

```
/* Quadratwurzel, Halbierungsverfahren, 14.08.92 */
/* Compileraufruf cc -o wurzel wurzel.c */

#define  EPS 0.00001
#define  MAX 100

#include <stdio.h>

void exit();

int main(int argc,char *argv[])

{
int i;
double a, b, c, m;

if (argc < 2) {
    puts("Radikand fehlt.");
    exit(-1);
}

/* Initialisierung */

i = 0;
sscanf(argv[1], "%lf", &c);
sscanf(argv[1], "%lf", &c);
a = 0;
b = c + 1;

/* Iteration */

while (b - a > EPS) {
    m = (a + b) / 2;
    if (m * m - c <= 0)
        a = m;
    else
        b = m;

/* Begrenzung der Anzahl der Iterationen */

    i++;
    if (i > MAX) {
         puts("Zuviele Iterationen! Ungenau!");
         break;
    }
}

/* Ausgabe und Ende */

printf("Die Wurzel aus %lf ist %lf\n", c, m);
printf("Anzahl der Iterationen: %d\n", i);
exit(0);
}
```

Quelle 1.54 : C-Programm zur iterativen Berechnung der Quadratwurzel

Die Funktion, die iterativ aufgerufen wird, ist die Mittelwertbildung von a und b; es lohnt sich nicht, sie auch programmtechnisch als selbständige Funktion zu definieren, aber das kann in anderen Aufgaben anders sein.

1.4.7 Rekursiver Aufruf einer Funktion

Bei einer **Rekursion** ruft eine Funktion sich selbst auf. Das ist etwas schwierig vorzustellen und nicht in allen Programmiersprachen erlaubt. Die Nähe zum Zirkelschluß ist nicht geheuer. Es gibt aber Probleme, die ursprünglich rekursiv sind und sich durch eine Rekursion elegant programmieren lassen. Eine **Zirkeldefinition** ist eine Definition eines Begriffes, die diesen selbst in der Definition enthält, damit es nicht sofort auffällt, gegebenenfalls um einige Ecken herum. Ein **Zirkelschluss** ist eine Folgerung, die Teile der zu beweisenden Aussage bereits zur Voraussetzung hat. Bei einer Rekursion hingegen

- wiederholt sich die Ausgangslage nie,
- wird eine Abbruchbedingung nach endlich vielen Schritten erfüllt, d. h. die Rekursionstiefe ist begrenzt.

In dem Buch von ROBERT SEDGEWICK findet sich Näheres zu diesem Thema, mit Programmbeispielen. Im ersten Band der *Informatik* von FRIEDRICH L. BAUER und GERHARD GOOS wird die Rekursion allgemeiner abgehandelt.

Zwei Beispiele sollen die Rekursion veranschaulichen. Das erste Programm berechnet den größten gemeinsamen Teiler (ggT) zweier ganzer Zahlen nach dem Algorithmus von EUKLID. Das zweite ermittelt rekursiv die Fakultät einer Zahl, was man anders vielleicht einfacher erledigen könnte.

```
/* Groesster gemeinsamer Teiler, Euklid, rekursiv */
/* Compileraufruf cc -o ggtr ggtr.c */

#include <stdio.h>

int ggt();

int main(int argc,char *argv[])

{
int x, y;

sscanf(argv[1], "%d", &x); sscanf(argv[2], "%d", &y);
printf("Der GGT von %d und %d ist %d.\n", x, y, ggt(x, y));
return 0;
}

/* Funktion ggt() */

int ggt(int a,int b)
{
if (a == b) return a;
else if (a > b) return(ggt(a - b, b));
else return(ggt(a, b - a));
}
```

Quelle 1.55 : C-Programm Größter gemeinsamer Teiler (ggT) nach Euklid, rekursiv

Im folgenden Programm ist außer der Rekursivität die Verwendung der Bedingten Bewertung interessant, die den Code verkürzt.

```
/* Rekursive Berechnung von Fakultaeten */

#include <stdio.h>

int main()
{
    int n;
    puts("\nWert eingeben, Ende mit CTRL-D");
    while (scanf("%d", &n) != EOF)
        printf("\n%d Fakultaet ist %d.\n\n", n, fak(n));
```

```
    return 0;
}

/* funktion fak() */

int fak(int n)
{
    return(n <= 1 ? 1 : n * fak(n - 1));
}
```

Quelle 1.56 : C-Programm zur rekursiven Berechnung der Fakultät

Weitere rekursiv lösbare Aufgaben sind die Türme von Hanoi und Quicksort. Rekursive Probleme lassen sich auch iterativ lösen. Das kann sogar schneller gehen, aber die Eleganz bleibt auf der Strecke.

Da in C auch das Hauptprogramm `main()` eine Funktion ist, die auf gleicher Stufe mit allen anderen Funktionen steht, kann es sich selbst aufrufen:

```
/* Experimentelles Programm mit Selbstaufruf von main() */

#include <stdio.h>

int main()
{
puts("Selbstaufruf von main()");
main();
return(13);
}
```

Quelle 1.57 : C-Programm, in dem main() sich selbst aufruft

Das Programm wird von `lint(1)` nicht beanstandet, einwandfrei kompiliert und läuft, bis der Speicher platzt, da die Rekursionstiefe nicht begrenzt ist (Abbruch mit break). Allerdings ist ein Selbstaufruf von `main()` ungebräuchlich.

1.4.8 Assemblerroutinen

Auf die Assemblerprogrammierung wurde in Abschnitt 1.1.4 *Programmiersprachen* auf Seite 7 bereits eingegangen. Da das Schreiben von Programmen in Assembler mühsam ist und die Programme nicht portierbar sind, läßt man nach Möglichkeit die Finger davon. Es kann jedoch zweckmäßig sein, einfache, kurze Funktionen auf Assembler umzustellen. Einmal kann man so unmittelbar auf die Hardware zugreifen, beispielsweise in Anwendungen zum Messen und Regeln, zum anderen zur Beschleunigung oft wiederholter Funktionen.

```
/* fakul.c Berechnung von Fakultaeten */

/* Die Grenze fuer END liegt in der Segmentgroesse */
/* bis 260 werden alle Werte in einem Array gespeichert,
   darueber wird Wert fuer Wert berechnet und ausgegeben */
/* Ziffern in Neunergruppen, nutzt long aus */

#define END  260
#define MAX 1023
#define DEF   16
#define GRP   58
#define GMX  245

/* GRP muss in aadd.asm eingetragen werden */
/* GMX muss in laadd.asm eingetragen werden */

#include <stdio.h>
```

```
unsigned long f[END + 1][GRP];          /* global */

void add(unsigned long *, unsigned long *);
void exit(int);
long time(long *);

/* Assemblerfunktionen zur Beschleunigung &/

extern void aadd(unsigned long *, unsigned long *);
extern void lshift(unsigned long *);

/* Hauptprogramm */

int main(int argc, char *argv[])

{
int e, i, j, k, r, s, flag, ende, max = DEF;
unsigned long x[GRP];
unsigned long *z;
long z1, z2, z3;

/* Auswertung der Kommandozeile */

if (argc > 1) {
        sscanf(*(argv + 1), "%d", &max);
        max = (max < 0) ? -max : max;
        if (max > MAX) {
           printf("\nZahl zu gross! Maximal %d\n", MAX);
           exit(1);
        }
}

ende = (max > END) ? END : max;

time(&z1);                        /* Zeit holen */

/* Rechnung */

**f = (unsigned long)1;

for (i = 1; i <= ende; i++) {
        for (j = 0; j < GRP; j++)    /* x nullsetzen */
                *(x + j) = 0;
        k = i/4;
        for (j = 1; j <= k; j++) {   /* addieren */
                aadd(x, *(f + i - 1));
        }
        lshift(x);
        lshift(x);
        for (k = 0; k < (i % 4); k++)
                aadd(x, *(f + i - 1));
        for (j = 0; j < GRP; j++) {  /* zurueckschreiben */
                *(*(f + i) + j) = *(x + j);

/* *(*(f + i) + j) ist dasselbe wie f[i][j] */

        }
}

time(&z2);                        /* Zeit holen */

/* Ausgabe, fuehrende Nullen unterdrueckt */

printf("\n\tFakultaeten von 0 bis %4d\n", max);

for (i = 0; i <= ende; i++) {
```

```
        flag = 0;
        printf("\n\t%4d ! =     ", i);
        for (j = GRP - 1; j >= 0; j-) {
            if (!(*(*(f + i) + j)) && !flag);
            else
                if (!flag) {
                    printf("%9lu ", *(*(f + i) + j));
                    flag = 1;
                }
                else
                    printf("%09lu ", *(*(f + i) + j));
        }
}

/* falls wir weitermachen wollen, muessen wir das Array
   f[261][58] umfunktionieren in f[2][*].
   In f[0] steht vorige Fakultaet, in f[1] wird addiert. */

if (max > END) {

/* f[0] einrichten */

        e = GMX;                /* kleiner 7296 */

        for (j = 0; j < e; j++)
                *(*f + j) = 0;  /* f[0] nullsetzen */

        aadd(f[0], f[END]);     /* vorige Fak. addieren */

/* Rechnung wie gehabt */

        r = 0; s = e;

        for (i = END + 1; i <= max; i++) {
                for (j = 0; j < e; j++)     /* f[1] nullen */
                        *(*(f + s) + j) = 0;

                k = i/4;
                for (j = 1; j <= k; j++) {  /* addieren */
                        laadd(*(f + s), *(f + r));
                }
                lshift(*(f + s));
                lshift(*(f + s));
                for (k = 0; k < (i % 4); k++)
                        laadd(*(f + s), *(f + r));

                flag = 0;               /* f[1] anzeigen */
                printf("\n\n\t%4d ! =     ", i);
                for (j = e - 1; j >= 0; j-) {
                   if (!(*(*(f + s) + j)) && !flag);
                   else
                      if (!flag) {
                          printf("%9lu ", *(*(f + s) + j));
                          flag = 1;
                      }
                      else
                          printf("%09lu ", *(*(f + s) + j));
                }
        r = (r > 0) ? 0 : e; /* f[1] wird das naechste f[0] */
        s = (s > 0) ? 0 : e;
        }
}

/* Ende Weitermachen */

/* Anzahl der Stellen von max! */
```

```
if (max > END) {
        ende = r; j = GMX - 1;
}
else {
        j = GRP - 1;
}

        flag = 0;
        for (; j >= 0; j-) {
                if (!(*(*(f + ende) + j)) && !flag)
                        ;
                else
                    if (!flag) {
                        unsigned long z = 10;
                        flag = 1;
                        for (i = 1; i < 9; i++) {
                            if (*(*(f + ende) + j) / z) {
                                        flag++;
                                        z *= 10;
                            }
                            else
                                break;
                        }
                    }
                    else
                        flag += 9;
        }

time(&z3);                        /* Zeit holen */

printf("\n\n\tZahl %4d ! hat %4d Stellen.\n", max, flag);

if (max > END)
  printf("\tRechnung+Ausgabe brauchten %4ld s.\n", z3 - z1);
else {
  printf("\tDie Rechnung brauchte   %4ld s.\n", z2 - z1);
  printf("\tDie Ausgabe brauchte    %4ld s.\n", z3 - z2);
}

return 0;
}
```

Quelle 1.58 : C-Programm zur Berechnung von Fakultäten

Das vorstehende Beispiel mit Microsoft Quick C und Quick Assembler für den IBM-PC bietet einen einfachen Einstieg in die Assemblerprogrammierung, da das große Programm nach wie vor in einer höheren Sprache abgefaßt ist. Das Beispiel ist in einer zweiten Hinsicht interessant. Auf einer 32-Bit-Maschine liegt die größte vorzeichenlose Ganzzahl etwas über 4 Milliarden. Damit kommen wir nicht weit, denn es ist bereits:

$$13! = 6227020800 \tag{1.3}$$

Wir stellen unsere Ergebnisse dar durch ein Array von langen Ganzzahlen, und zwar packen wir immer neun Dezimalstellen in ein Array-Element:

```
unsigned long f[END + 1][GRP]
```

Bei asymmetrischen Verschlüsselungsverfahren braucht man große Zahlen. Die Arithmetik zu diesem Datentyp müssen wir selbst schreiben. Dazu ersetzen wir die eigentlich bei der Berechnung von Fakultäten erforderliche Multiplikation durch die Addition. Diese beschleunigen wir durch Einsatz einer Assemblerfunktion `aadd()`:

```
COMMENT +
C-Funktion aadd() in MS-Assembler, die ein
```

```
                Array of unsigned long in ein zweites Array
                addiert, Parameter Pointer auf die Arrays.
                +

                .MODEL  small,c
                .CODE
grp             EQU     4 * 58          ; siehe C-Programm
milliarde       DD      1000000000

; fuer laadd() obige Zeile austauschen
; grp           EQU     4 * 245         ; siehe C-Programm


aadd            PROC    USES SI, y:PTR DWORD, g:PTR DWORD

                sub     cx,cx
                sub     si,si
                clc
; for-Schleife nachbilden
for1:
; aktuelles Element in den Akku holen, long = 4 Bytes!
                mov     bx,y
                mov     ax,WORD PTR [bx+si][0]
                mov     dx,WORD PTR [bx+si][2]
; Uebertrag zu Akku addieren
                add     ax,cx
                adc     dx,0
; vorige Fakultaet zu Akku addieren, Uebertrag beachten
                mov     bx,g
                add     ax,WORD PTR [bx+si][0]
                adc     dx,WORD PTR [bx+si][2]
; Summe durch 10 hoch 9 dividieren, Quotient ergibt
; Uebertrag ins naechste Element des Arrays, Rest
: ergibt aktuelles Element.
; zweite for-Schleife:
                sub     cx,cx
for2:
                cmp     dx,WORD PTR milliarde[2]
                jl      SHORT fertig
                sub     ax,WORD PTR milliarde[0]
                sbb     dx,WORD PTR milliarde[2]
                inc     cx
                jmp     SHORT for2
fertig:
; Rest zurueckschreiben in aktuelles Array
                mov     bx,y
                mov     WORD PTR [bx+si][0],ax
                mov     WORD PTR [bx+si][2],dx
; Schleifenzaehler um 4 (long!) erhoehen
                add     si,4
; Ruecksprungbedingung
                cmp     si,grp
                je      SHORT done
                jmp     SHORT for1
; Ende der Funktion
done:
                ret
aadd            ENDP
                END
```

Quelle 1.59 : Assemblerfunktion 1 zur Addition von Feldern

Die Fakultäten werden berechnet, gespeichert und zum Schluß zusammen ausgegeben. So können wir die Rechenzeit von der Ausgabezeit trennen. Es zeigt sich, daß die Rechenzeit bei Ganzzahl-Arithmetik gegenüber der Bildschirmausgabe keine Rolle spielt.

Auf diesem Weg kommen wir bis in die Gegend von 260!, dann ist ein Speichersegement (64 KByte) unter PC-DOS voll. Wir können nicht mehr alle Ergebnisse speichern, sondern nur die vorangegangene und die laufende Fakultät. Sowie ein Ergebnis vorliegt, wird es ausgegeben. Die Assemblerfunktion `laadd()` zur Addition unterscheidet sich in einer Zeile am Anfang. Die im Programm vorgesehene Grenze $MAX = 1023$ ist noch nicht die durch das Speichersegment bestimmte Grenze, sondern willkürlich. Irgendwann erheben sich Zweifel am Sinn großer Zahlen. Selbst als Tapetenmuster wirken sie etwas eintönig.

1.4.9 Memo Funktionen

- C-Programme sind aus gleichberechtigten Funktionen aufgebaut. Zu diesen gehört auch `main()`.
- Eine Funktion übernimmt bei ihrem Aufruf einen festgelegten Satz von Parametern oder Argumenten. Der Satz beim Aufruf muß mit dem Satz bei der Definition nach Anzahl, Typ und Reihenfolge übereinstimmen wie Stecker und Kupplung einer elektrischen Steckverbindung.
- Bei der Parameterübergabe by value arbeitet die Funktion mit Kopien der übergebenen Parameter, kann also die Originalwerte nicht verändern.
- Bei der Parameterübergabe by reference erfährt die Funktion die Adressen (Pointer) der Originalwerte und kann diese verändern. Das ist gefährlicher, aber manchmal gewollt. Beispiel: `scanf()`.
- Auch die Funktion `main()` kann Argumente übernehmen, und zwar aus der Kommandozeile. Die Argumente stehen in einem Array of Strings (Argumentvektor).
- Es gibt auch Funktionen wie `printf()`, die eine von Aufruf zu Aufruf wechselnde Anzahl von Argumenten übernehmen. Der Mechanismus ist an einige Voraussetzungen gebunden.
- Eine Funktion gibt keinen oder genau einen Wert als Ergebnis an die aufrufende Funktion zurück. Dieser Wert kann ein ein Pointer sein.
- In C darf eine Funktion sich selbst aufrufen (rekursiver Aufruf).
- Assemblerfunktionen innerhalb eines C-Programms können den Ablauf beschleunigen. Einfacher wird das Programm dadurch nicht.

1.4.10 Übung Funktionen

Jetzt verfügen Sie über die Kenntnisse, die zum Schreiben einfacher C-Programme notwendig sind. Schreiben das Programm zur Weganalyse, aufbauend auf der Aufgabenanalyse und der Datenstruktur, die Sie bereits erarbeitet haben. Falls Sie sich an den Vokabeltrainer wagen wollen, reduzieren Sie die Aufgabe zunächst auf ein Minimum, sonst werden Sie nicht fertig damit.

1.5 Funktions-Bibliotheken

1.5.1 Zweck und Aufbau

Eine Funktion kann auf drei Wegen mit einem C-Hauptprogramm `main()` verbunden werden:

- Die Funktion steht in derselben Datei wie `main()` und wird daher gemeinsam kompiliert. Sie muss wie `main()` in C geschrieben sein, mehrsprachige Compiler gibt es nicht.

- Die Funktion steht – unter Umständen mit weiteren Funktionen – in einer eigenen Datei, das getrennt kompiliert und beim Linken zu `main()` gebunden wird. Dabei werden alle Funktionen dieser Datei zu `main()` gebunden, ob sie gebraucht werden oder nicht. Wegen der getrennten Compilierung darf die Datei in einer anderen Programmiersprache geschrieben, muß aber für dieselbe Maschine kompiliert sein.
- Die getrennt kompilierte Funktion steht zusammen mit weiteren in einer Bibliothek und wird beim Linken zu `main()` gebunden. Dabei wählt der Linker nur die Funktionen aus der Bibliothek aus, die in `main()` gebraucht werden. Man kann also viele Funktionen in einer Bibliothek zusammenfassen, ohne befürchten zu müssen, seine Programme mit Ballast zu befrachten. Die Bibliothek kann auf Quellfiles unterschiedlicher Programmiersprachen zurückgehen. Sie müssen nur für dasselbe System kompiliert worden sein; es macht keinen Sinn und ist auch nicht möglich, Funktionen für UNIX und PC-DOS in einer Bibliothek zu vereinigen.

Das Erzeugen einer Bibliothek auf UNIX-Systemen wurde bereits im Abschnitt 1.2.6 *Bibliotheken, Archive* auf Seite 25 im Rahmen der Programmer's Workbench erläutert. Im folgenden geht es um die Verwendung von Bibliotheken.

1.5.2 Standardbibliothek

1.5.2.1 Übersicht

Standardfunktionen sind die Funktionen, die als **Standardbibliothek** zusammen mit dem Compiler geliefert werden. Sie sind im strengen Sinn nicht Bestandteil der Programmiersprache – das bedeutet, daß sie ersetzbar sind – aber der ANSI-Standard führt eine minimale Standardbibliothek auf. Ohne sie könnte man kaum ein Programm in C schreiben. Der Reichtum der Standardbibliothek ist eine Stärke von C; in ihr steckt viel Arbeit, die andere Programmierer vor und für uns erledigt haben. Die Systemaufrufe (Sektion 2) gehören dagegen nicht zur Standardbibliothek (Sektion 3), sondern zum Betriebssystem. Und Shell-Kommandos sind eine Sache der Shell (Sektion 1). Diese Unterscheidung spielt eine Rolle, wenn Programme portiert werden. Soweit möglich sind Standardfunktionen zu verwenden, da sie Eigenheiten des Systems verdecken.

Die mit dem C-Compiler eines UNIX-Systems mitgelieferte Standardbibliothek wird im Referenz-Handbuch unter `intro(3)` vorgestellt und umfaßt mehrere Teile:

- die Standard-C-Bibliothek, meist gekoppelt mit der Standard-Input-Output-Bibliothek, den Netzfunktionen und den Systemaufrufen (weil sie zusammen gebraucht werden),
- die mathematische Bibliothek,
- gegebenenfalls eine grafische Bibliothek,
- gegebenenfalls eine Bibliothek mit Funktionen zum Messen und Regeln,
- gegebenenfalls Datenbankfunktionen und weitere Spezialitäten.

Außer Funktionen enthält sie Include-Dateien mit Definitionen und Makros, die von den Funktionen benötigt werden, im UNIX-Datei-System aber in einem anderen Verzeichnis (`/usr/include`) liegen als die Funktions-Bibliotheken (`/lib` und `/usr/lib`).

1.5.2.2 Standard-C-Bibliothek

Die Standard-C-Bibliothek `/lib/libc.a` wird vom C-Compilertreiber `cc(1)` eines UNIX-Systems aufgerufen und braucht daher nicht als Option mitgegeben zu werden. Für einen getrennten Linker-Aufruf lautet die Option `-lc`. Mit dem Kommando:

```
ar -t /lib/libc.a
```

schauen Sie sich das Inhaltsverzeichnis der Bibliothek an. Außer bekannten Funktionen wie `printf()` und Systemaufrufen wie `stat(2)` werden Sie viele Unbekannte treffen. Auskunft über diese erhalten Sie mittels der man-Seiten, beispielsweise:

```
man ruserok
man insque
```

sofern die Funktionen zum Gebrauch durch Programmierer und nicht etwa nur für interne Zwecke bestimmt sind.

Input/Output Für die Ein- und Ausgabe stehen in C keine Operatoren zur Verfügung, sondern nur die **Systemaufrufe** des Betriebssystems (unter UNIX `open(2)`, `write(2)`, `read(2)` usw.) und die **Standardfunktionen** aus der zum Compiler gehörenden Bibliothek. In der Regel sind die Funktionen vorzuziehen, da die Programme dann leichter auf andere Systeme übertragen werden können. Die Eigenheiten der Systeme werden durch die Bibliothek verdeckt. In diesem Fall ist im Programmkopf stets die Include-Datei `stdio.h` einzubinden:

```
#include <stdio.h>
```

Diese Zeile ist fast in jedem C-Programm zu finden. In der Standardbibliothek stehen rund 40 Funktionen zur Ein- und Ausgabe bereit, von denen die bekanntesten `printf(3)` zur formatierten Ausgabe nach `stdout` und `scanf(3)` zur formatierten Eingabe von `stdin` sind.

Stringfunktionen Strings sind in C Arrays of Characters, abgeschlossen mit dem ASCII-Zeichen Nr. 0, also nichts Besonderes. Trotzdem machen sie – wie in vielen Programmiersprachen – Schwierigkeiten, wenn man ihre Syntax nicht beachtet. In der Standard-C-Bibliothek steht eine reiche Auswahl von Stringfunktionen bereit. Gewarnt wird vor der Funktion `gets(3)`, die einen String von `stdin` in ein Array einliest und dabei einen Überlauf verursachen kann. Grundsätzlich soll man beim Arbeiten mit Strings den Fall berücksichtigen, dass ein String gelegentlich länger ist als der Puffer, in den er geschrieben werden soll.

Da ein **String** – wie jedes Array – keinen Wert hat, kann er nicht per Zuweisung einer Stringvariablen zugewiesen werden. Man muß vielmehr mit den Standard-Stringfunktionen arbeiten oder sich selbst um die einzelnen Elemente des Arrays kümmern. Die Stringfunktionen erwarten die Include-Datei `string.h`. Hier ein kurzes C-Programm zur Stringmanipulation mittels Systemaufrufen und Standardfunktionen:

```
/* Programm fuer Stringmanipulation */

#define TEXT "textfile"

#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>
#include <string.h>

char buffer[80] = "Dies ist ein langer Teststring. Hallo!";

int main()
{
    char    x, zeile[80];
    int i;
    int fildes;
    FILE * fp;

    /* Systemaufrufe und Filedeskriptoren */

    fildes = open(TEXT, O_RDWR);
    if (fildes == -1)
        puts("open schiefgegangen.");
    write(fildes, buffer, 20);
    lseek(fildes, (long)0, SEEK_SET);
    read(fildes, zeile, 12);
    write(1, zeile, 12);
    close(fildes);
```

```
    /* Standardfunktionen und Filepointer */

    fp = fopen(TEXT, "w");
    for (i = 0; i < 30; i++)
        fputc(buffer[i], fp);
    fclose(fp);
    fp = fopen(TEXT, "r");
    for (i = 0; i < 30; i++)
        zeile[i] = fgetc(fp);
    putchar('\n');
    for (i = 0; i < 30; i++)
        putchar(zeile[i]);

    /* Stringfunktionen */

    strcpy(zeile, buffer);
    printf("\n%s", zeile);
    putstf("\n\nBitte eine Zeile eingeben:");
    gets(zeile);
    puts(zeile);
    strcat(zeile, " Prima!");
    puts(zeile);
    printf(zeile);
}
```

Quelle 1.60 : C-Programm zur Stringverarbeitung

Internet-Funktionen Eine Übersicht über diese Funktionen findet sich in `intro(3N)`. Beispiele sind Funktionen zur Verarbeitung von Netzadressen, Protokolleinträgen, Remote Procedure Calls, zum Mounten ferner Datei-Systeme, zur Verwaltung von Benutzern und Passwörtern im Netz. Geht über den Rahmen dieses Textes hinaus. Falls Sie sich ein eigenes Programm `telnet` oder `ftp` schreiben wollten, müßten Sie hier tiefer einsteigen.

1.5.2.3 Standard-Mathematik-Bibliothek

Die **Standard-Mathematik-Bibliothek** wird vom C-Compilertreiber nicht automatisch aufgerufen – anders als in FORTRAN. Für C ist die Option `-lm` hinzuzufügen:

```
cc -o myprogram myprogram.c -lm
```

Ferner muß im Programmkopf die Zeile

```
#include <math.h>
```

stehen. Dann verfügt man über Logarithmus, Wurzel, Potenz, trigonometrische und hyperbolische Funktionen. Weiteres siehe `math(5)`.

Eigentlich sollte man bei diesen Funktionen den zugrunde liegenden Algorithmus und seine Programmierung kennen, da jedes numerische Verfahren und erst recht seine Umsetzung in ein Programm Grenzen haben, aber das Referenz-Handbuch beschränkt sich unter `trig(3)` usw. auf die Syntax der Funktionen. Ein Beispiel für die Verwendung der mathematischen Bibliothek:

```
/* Potenz x hoch y; mathematische Funktionen; 22.12.92 */
/* zu compilieren mit cc -o potenz potenz.c -lm */
/* Aufruf: potenz x y */

#define  EPSILON  0.00001
#include <stdio.h>
#include <math.h>

double pow(), floor();

int main(int argc,char *argv[])
```

```
{
double x, y, z;

if (argc < 3) {
    puts("Zuwenig Argumente");
    return(-1);
}

/* Umwandlung Kommandozeilenargumente */

sscanf(argv[1], "%lf", &x);
sscanf(argv[2], "%lf", &y);

/* Aufruf Funktionen pow(), floor(), Sektion 3M */
/* wegen Fallunterscheidungen nachlesen! */

if ((x < 0 ? -x : x) < EPSILON) {
    if (y > 0) x = 0;
    else {
        puts("Bei x = 0 muss y positiv sein.");
        return(-1);
    }
}
else {
    if (x < 0) y = floor(y);
    }

z = pow(x, y);

/* Ausgabe */

printf("%lf hoch %lf = %lf\n", x, y, z);

return 0;
}
```

Quelle 1.61 : C-Programm mit mathematischen Funktionen

Der `lint(1)` gibt bei diesem Programm eine längere Liste von Warnungen aus, die daher rühren, daß in `<math.h>` viele Funktionen deklariert werden, die im Programm nicht auftauchen. Das geht aber in Ordnung.

1.5.2.4 Standard-Grafik-Bibliothek

Zu manchen Compilern gehört auch eine Sammlung von Grafikfunktionen. Da es hierfür noch keinen Standard gibt und Grafik eng an die Hardware gebunden ist, verzichten wir auf eine Darstellung. Auf einer UNIX-Anlage findet man sie in `/usr/lib/plot`. Die Bibliothek enthält Funktionen zum Setzen von Punkten, Ziehen von Linien, zur Umwandlung von Koordinaten und ähnliche Dinge. Leider nicht standardisiert, sonst gäbe es nicht Starbase, GKS, OpenGL, PHIGS, Uniras ...

1.5.2.5 Weitere Teile der Standardbibliothek

Die nicht zur Standard-C-Bibliothek gehörenden `curses(3)`-Funktionen aus `/usr/lib/libcurses.a` ermöglichen die weitergehende Gestaltung eines alphanumerischen Bildschirms. In diesem Fall ist die `curses(3)`-Bibliothek beim Compileraufruf zu nennen:

```
cc .... -lcurses
```

Vergißt man die Nennung, weiß der Compiler mit den Namen der `curses(3)`-Funktionen nichts anzufangen und meldet sich mit der Fehleranzeige `unsatisfied symbols`.

Bei Verwendung von `curses(3)`-Funktionen ist die Include-Datei `curses.h` in das Programm aufzunehmen, das `stdio.h` einschließt.

1.5.3 Xlib, Xt und Xm (X Window System)

Programme, die von dem X Window System Gebrauch machen wollen, greifen auf unterster Ebene auf Funktionen der **Xlib-Bibliothek** zurück. Die Xlib stellt für jede Möglichkeit des X-Protokolls eine C-Funktion bereit; sie ist die Schnittstelle zwischen C-Programmen und dem X-Protokoll.

Auf nächsthöherer Ebene werden Funktionen einer Toolbox wie der **Xt-Bibliothek** definiert, die ihrerseits auf der Xlib aufsetzt. Die Xt-Funktionen werden auch als Intrinsics bezeichnet. Sie kennen z. B. Widgets, das sind Window Gadgets[32] oder Objekte (im Sinne von C++) des Client-Programms. Zu einem **Widget** gehören sein Fenster, sein Aussehen (look), sein Verhalten (feel) und ein Satz von Methoden, die sein Verhalten realisieren. Ein Menü oder ein anklickbarer Druckknopf ist ein Widget.

Die dritte Schicht bilden Bibliotheken wie der Motif Toolkit Xm, der eine Menge nach einheitlichen Regeln gebauter Widgets zur Verfügung stellt. Während Xt nur abstrakte Fenster und Menus kennt, legt Xm fest, wie ein (Motif-)Fenster oder -Menü aussieht und wie es sich verhält. Während der Quellcode von Xlib und Xt veröffentlicht ist, kostet die Xm eine Kleinigkeit. Ein Programmierer versucht immer, mit der höchsten Bibliothek zu arbeiten, weil er sich dabei am wenigsten um Einzelheiten zu kümmern braucht.

1.5.4 NAG-Bibliothek

Die **NAG-Bibliothek** der Numerical Algorithms Group, Oxford soll hier als ein Beispiel für eine umfangreiche kommerzielle Bibliothek stehen, die bei vielen numerischen Aufgaben die Arbeit erleichtert. Die FORTRAN-Bibliothek umfaßt etwa 1200 Subroutinen, die C-Bibliothek etwa 250 Funktionen. Sie stammen aus folgenden Gebieten:

- Nullstellen, Extremwerte,
- Differential- und Integralgleichungen,
- Fourier-Transformation,
- Lineare Algebra,
- nichtlineare Gleichungen,
- Statistik,
- Näherungen, Interpolation, Ausgleichsrechnung,
- Zufallszahlen usw.

Näheres unter `http://www.nag.co.uk:80/numeric.html`.

1.5.5 Eigene Bibliotheken

Wir haben bereits in Abschnitt 1.2.6 *Bibliotheken, Archive* auf Seite 25 gelernt, eine eigene Programmbibliothek mittels des UNIX-Kommandos `ar(1)` herzustellen. Zunächst macht es Arbeit, seine Programmierergebnisse in eine Bibliothek einzuordnen, aber wenn man einmal einen Grundstock hat, zahlt es sich aus.

[32]Ein Gadget ist laut Wörterbuch ein geniales Dingsbums.

1.5.6 Speichermodelle (PC-DOS)

Unter UNIX gibt es keine Speichermodelle, infolgedessen auch nur eine Standardbibliothek. Unter PC-DOS hingegen ist die Speichersegmentierung zu beachten, d. h. die Unterteilung des Arbeitsspeichers in Segmente zu je 64 kByte, ein lästiges Überbleibsel aus uralten Zeiten. Die Adressierung der Speicherplätze ist unterschiedlich, je nachdem ob man sich nur innerhalb eines Segmentes oder im ganzen Arbeitsspeicher bewegt. Für jedes Speichermodell ist eine eigene Standardbibliothek vorhanden. Das Speichermodell wird gewählt durch:

- die Angabe einer Compiler-Option oder
- die Schlüsselwörter `near`, `far` oder `huge` im C-Programm (was unter UNIX-C zu einem Fehler führt)

Wird keine der beiden Möglichkeiten genutzt, nimmt der Compiler einen Default an, MS-Quick-C (`qcl`) beispielsweise das Modell `small`.

Das Modell `tiny` (nicht von allen Compilern unterstützt) packt Code, Daten und Stack in ein Segment; für die Adressen (Pointer) reichen 2 Bytes. Das gibt die schnellsten Programme, aber hinsichtlich des Umfangs von Programm und Daten ist man beschränkt.

Das Modell `small` packt Code und Daten in je ein Segment von 64 kByte. Damit lassen sich viele Aufgaben aus der PC-DOS-Welt bewältigen.

Das Modell `medium` stellt ein Segment für Daten und mehrere Segmente für Programmcode zur Verfügung, bis zur Grenze des freien Arbeitsspeichers. Typische Anwendungen sind längere Programme mit wenigen Daten.

Das Modell `compact` verhält sich umgekehrt wie `medium`: ein Segment für den Code, mehrere Segmente für die Daten. Geeignet für kurze Programme mit vielen Daten. Ein einzelnes Datenelement – ein Array beispielsweise – darf nicht größer als ein Segment sein.

Das Modell `large` läßt jeweils mehrere Segmente für Code und Daten zu, wobei wieder ein einzelnes Datenelement nicht größer als ein Segment sein darf.

Das Modell `huge` schließlich hebt auch diese letzte Beschränkung auf, aber die Beschränkung auf die Größe des freien Arbeitsspeichers bleibt, PC-DOS schwoppt nicht.

Die Schlüsselwörter `near`, `far` und `huge` in Verbindung mit Pointern oder Funktionen haben Vorrang vor dem vom Compiler benutzten Speichermodell. Bei `near` sind alle Adressen 16 Bits lang, bei `far` sind die Adressen 32 Bits lang, die Pointerarithmetik geht jedoch von 16 Bits aus, und bei `huge` schließlich läuft alles mit 32 Bits und entsprechend langsam ab. Falls Ihnen das zu kompliziert erscheint, steigen Sie einfach um auf UNIX.

1.5.7 Memo Bibliotheken

- Eine Bibliothek vereint eine Menge von Funktionen in einer einzigen Datei.
- Eine Bibliothek hat *nichts* mit Verschlüsseln oder Komprimieren zu tun.
- Beim Einbinden einer Bibliothek in den Kompiliervorgang werden genau die benötigten Funktionen ausgewählt und ins Programm eingebunden.
- Es gibt Standardbibliotheken, die zum Compiler gehören und von diesem automatisch herangezogen werden.
- Weiter gibt es Standardbibliotheken, die zum Compiler gehören, aber eigens über eine Option herangezogen werden müssen. Hierzu zählt die C-Standard-Mathematik-Bibliothek, die die Option `-lm` beim Compiler-Aufruf erfordert.
- Auf dem Markt oder im Netz findet sich eine Vielzahl von Bibliotheken, beispielsweise für numerische Aufgaben oder das X Window System.
- Man kann auch eigene Funktionen in Privatbibliotheken zusammenfassen. Das lohnt sich, wenn man längere Zeit für ein bestimmtes Thema programmiert.

1.5.8 Übung Bibliotheken

Fassen Sie die Funktionen des Weganalyse-Projektes außer `main()` in einer Privatbibliothek zusammen und binden sie diese beim Kompiliervorgang dazu.

1.6 Klassen

1.6.1 Warum C mit Klassen?

Objektorientiert oder prozedural ist nicht die Programmiersprache, sondern die Aufgabenanalyse. Sie führt auf Programmbausteine (Module), die entweder Objekte oder Prozeduren (Funktionen, Prozeduren, Subroutinen) sind. Erst an zweiter Stelle kommen dann die Programmiersprachen, die die eine oder andere Denkweise unterstützen. Man kann mit objektorientierten Sprachen prozedural aufgebaute Programme schreiben und mit prozeduralen Sprachen objektorientierte Programme. Da der Ausgangspunkt die Aufgabenanalyse ist, macht sich die Objektorientierung bei kleinen Programmen (wo es nichts zu analysieren gibt) nicht bemerkbar. C++ und Objective-C wurden entwickelt, um

- ein besseres C zu sein (dasselbe Ziel wie ANSI-C),
- die Datenabstraktion zu unterstützen,
- das objektorientierte Programmieren zu unterstützen.

Als erstes ein Hallo-Programm in C++ (mit Objektorientierung und Klassen ist da noch nichts zu machen):

```
/* Hallo, Welt; in C++ */

#include <iostream.h>            // anstelle stdio.h

int main()
{
char v[20];

cout « "Bitte Vornamen eingeben: ";
cin  » v;
cout « "Hallo, " « v « '\n';
return 0;
}
```

Quelle 1.62 : C++-Programm Hallo, Welt

Eine zweite Art des Kommentars (Zeilenkommentar) ist hinzugekommen. Der Operator << schreibt sein zweites Argument auf das erste, hier der Standard Output Stream `cout`. Der Operator >> schreibt sein erstes Argument, den Standard Input Stream, auf das zweite, den String `v`. Das Stream-Konzept zur Ein- und Ausgabe ist flexibler als das herkömmliche Datei-Konzept; für den Programmierer ist die andere Syntax wichtig (beachte: kein Formatstring! Der Operator weiß aufgrund der Typen, was er vor sich hat). In C++ gibt es eine Vielzahl solcher Verbesserungen oder Erweiterungen von C, aber sie sind nichts grundsätzlich Neues; sie erfordern kein Umdenken, sondern nur das Lesen des Referenzmaterials.

Obiges Programm `hallo.cpp` ist mit dem GNU `gcc` kompiliert 96 kB groß. Ein Assemblerprogramm, das dasselbe tut, belegt 120 Bytes. Die Speicherhersteller profitieren mit Sicherheit von der Objektorientierung.

1.6.2 Datenabstraktion, Klassenbegriff

In C ebenso wie in FORTRAN oder PASCAL beschreibt ein Datentyp eine Menge von Werten samt den zugehörigen Operationen. Die Datentypen sind durch den Compiler festgelegt, der Benutzer kann keine neuen Datentypen definieren.

Ein **abstrakter Datentyp** ist ein vom Benutzer definierter Typ, dessen Schnittstelle (Interface, Außenseite, Verbindung zum übrigen Programm) von seiner Implementierung (Implementation, interne Programmierung, Innenleben) getrennt ist, eine Black Box mit bestimmten nach außen sichtbaren Eigenschaften. **Klassen** sind in einer Programmiersprache formulierte Beschreibungen abstrakter Datentypen. **Objekte** sind Vertreter (Exemplare, Instanzen, Verwirklichungen) von Klassen. C-Typen und C++-Klassen sowie C-Variable

und C++-Objekte entsprechen sich. Aus Klassen lassen sich untergeordnete Klassen ableiten. Eine Klasse oder ein Objekt enthält **Daten** (data member) und Operationen auf diesen Daten. Die Operationen, die in den Klassen oder Objekten verwirklicht sind, heißen **Methoden** (member function, method). Objekte verkehren untereinander mittels Botschaften. Eine **Botschaft** (message, member function call) ist die Aufforderung an ein Objekt, eine seiner Methoden auszuführen, vergleichbar einem Funktionsaufruf in C.

Beispielsweise können wir eine Klasse *Komplexe Zahl* definieren, die als Daten zwei reelle Zahlenwerte (Realteil und Imaginrrteil) sowie als Methoden die Grundrechenarten für komplexe Zahlen enthält:

```
Klasse KOMPLEX
{
Daten: double realteil, imaginaerteil;
Methoden:
       KOMPLEX  Addiere(a: KOMPLEX, b: KOMPLEX);
       KOMPLEX  Subtrahiere(a: KOMPLEX, b:KOMPLEX);
       KOMPLEX  Multipliziere(a: KOMPLEX, b: KOMPLEX);
       KOMPLEX  Dividiere(a: KOMPLEX, b: KOMPLEX);
}
```

Mitglieder (Daten, Methoden) sind öffentlich (public) oder privat. **Public Members** sind vom übrigen Programm her zugänglich, sie bilden die Schnittstelle der Klasse und ihrer Objekte zur Umwelt. **Private Members** sind nur den Methoden der Klasse zugänglich. Public und private werden als **Member Access Specifier** bezeichnet. Meist sind die Daten privat, die Methoden teils privat, teils öffentlich. Mindestens eine Methode muß öffentlich sein (warum?). Eine besondere Methode - mit demselben Namen wie die Klasse - ist der **Constructor**, der zur Initialisierung dient. Diese Methode wird automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird. Ein Constructor kann public, protected oder private sein; er hat niemals einen Rückgabewert eines bestimmten Typs, auch nicht des Typs `void`.

Nun ein funktionsfähiges (wenn auch simples) Beispiel. Es rechnet die Zeiten von UTC nach MEZ um:

```
/* mez.cpp, Beispiel fuer den Gebrauch einer Klasse
   in Anlehnung an Deitel + Deitel, S. 601 */

#include <iostream.h>          // fuer Ein- und Ausgabe

class TIME {                   // Definition einer Klasse

public:                        // nach aussen sichtbar,
                               // Methoden
    TIME();                    // Default Constructor,
                               // Initialisierung
    void Settime(int, int);    // h, m in UTC setzen
    void Gettime();            // UTC einlesen von stdin
    void Printmez();           // MEZ ausgeben

private:                       // nicht nach aussen sichtbar,
                               // Daten
    int hour;                  // 0 - 23
    int minute;                // 0 - 59
    int hin, min;              // Eingabe von stdin
};

// Definition der Methoden

// Initialisierung mittels Constructor

TIME::TIME() { hour = minute = 0; }

// Zeit in UTC eingeben, pruefen
```

```
void TIME::Settime(int h, int m)
{
    hour = (h >= 0 && h < 23) ? h + 1 : 0;
                                 // UTC nach MEZ
    minute = (m >= 0 && m < 60) ? m : 0;
}

// Zeit in UTC von stdin einlesen

void TIME::Gettime()
{
    cout « "Stunde eingeben: ";
    cin  » hin;
    cout « "Minuten eingeben: ";
    cin  » min;
    cout » "Vielen Dank" » endl;

    TIME::Settime(hin, min);  // Umrechnung
}

// MEZ ausgeben

void TIME::Printmez()
{
    cout « (hour < 10 ? "0" : "") « hour
         « ’:’
         « (minute < 10 ? "0" : "") « minute
         « endl;
}

// Hauptprogramm (Rahmen- oder Treiberprogramm)

int main()
{
    TIME t;            // Erzeugung des Objektes t

    cout « "\nDie Anfangszeit ist ";
    t.Printmez();      // Aufruf einer oeff. Methode

    t.Settime(13, 27);
    cout « "Neue Zeit ist ";
    t.Printmez();

    t.Gettime();
    cout « "Ihre Zeit ist: ";
    t.Printmez();

    t.Settime(99, 99);         // ungueltige Werte
    cout « "Fehlerhafte Eingabe fuehrt zu ";
    t.Printmez();

    cout « endl;       // endline stream manipulator
    return 0;
}
```

Quelle 1.63 : C++-Programm zur Umrechnung von UTC nach MEZ

1.6.3 Klassenhierarchie, abstrakte Klassen, Vererbung

Objektorientiertes Programmieren besteht im Programmieren einer Menge von Klassen, deren zughörige Objekte den Programmablauf verwirklichen. Das folgende Programm zeigt, wie aus einer Basisklasse weitere Klassen abgeleitet werden, die die public und protected members erben. Von einer **abstrakten Klasse** können nur weitere Klassen abgeleitet, jedoch keine Objekte gebildet werden. Eine abstrakte Klasse muß mindestens eine rein **virtu-**

elle Funktion enthalten, die nirgends definiert wird. Sie ist ein Platzhalter, der erst in einer abgeleiteten Klasse einen Inhalt bekommt.

```
/* geof.C, Beispiel fuer Klassen und Vererbung
   - geometrische Formen -
   Compileraufruf (HP): CC -o geof geof.C
*/

#define PI 3.14159                  // symbolische Konstante

#include <iostream.h>               // fuer Ein- und Ausgabe
#include <string.h>                 // wegen strcmp()
#include <stdlib.h>                 // wegen exit()

void exit(int);                     // Prototyp Systemaufruf

class Form {                        // abstrakte Basisklasse

public:                             // nach aussen sichtbar
    virtual void lesen() = 0;       // reine virt. Funktionen
    virtual void schreiben() = 0;

protected:                          // public fuer abg. Klasse,
                                    // ansonsten private
private:                            // nach aussen unsichtbar
};

class Flaeche : public Form {       // abgel. abstr. Klasse

public:
    Flaeche() {u = i = 0;}          // Constructor
    void schreiben()
        {cout « "Umfang = " « u « endl;
         cout « "Inhalt = " « i « endl;}

protected:
    double u, i;
    virtual double umfang(double, double) = 0;
                                    // rein virtuelle Fkt.
    virtual double inhalt(double, double) = 0;

private:
};

class Koerper : public Form {
                                    // abgel. abstr. Klasse

public:
    Koerper() {f = v = 0;}          // Constructor
    void schreiben()
        {cout « "Oberflaeche = " « f « endl;
         cout « "Volumen = " « v « endl;}

protected:
    double f, v;
    virtual double flaeche(double, double, double) = 0;
                                    // rein virtuelle Fkt.
    virtual double volumen(double, double, double) = 0;

private:
};

class Kreis : public Flaeche {      // abgel. konkr. Klasse

public:
    Kreis() {a = x = y = 0;}        // Constructor
```

```
    void lesen()
        {cout « "Radius: "; cin » a;
         u = umfang(a, a);
         i = inhalt(a, a);}

protected:

private:
    double a, x, y;
    double umfang(double x, double y) {return(PI * (x + y));}
    double inhalt(double x, double y) {return(PI * x * y);}
};

class Rechteck : public Flaeche { // abgel. konkr. Klasse

public:
    Rechteck() {a = b = x = y = 0;}
    void lesen() {cout « "Laenge: "; cin » a;
                  cout « "Breite: "; cin » b;
                  u = umfang(a, b);
                  i = inhalt(a, b);}

protected:
    double umfang(double x, double y) {return(2 * (x + y));}
    double inhalt(double x, double y) {return(x * y);}

private:
    double a, b, x, y;
};

class Quadrat : public Rechteck { // abgel. konkr. Klasse

public:
    Quadrat() {a = 0;}            // Constructor
    void lesen() {cout « "Laenge: "; cin » a;
                  u = umfang(a, a);
                  i = inhalt(a, a);}

protected:

private:
    double a;
};

class Kugel : public Koerper {    // abgel. konkr. Klasse

public:
    Kugel() {a = x = y = z = 0;}  // Constructor
    void lesen()
        {cout « "Radius: "; cin » a;
         f = flaeche(a, a, a);
         v = volumen(a, a, a);}

protected:

private:
    double a, x, y, z;
    double flaeche(double x, double y, double z)
        {return(2 * PI * x * (y + z));}
    double volumen(double x, double y, double z)
        {return(4 * PI * x * y * z / 3);}
};

class Quader : public Koerper {   // abgel. konkr. Klasse

public:
```

```
    Quader() {a = b = c = x = y = z = 0;} // Constructor
    void lesen()
        {cout « "Laenge: "; cin » a;
         cout « "Breite: "; cin » b;
         cout « "Hoehe:  "; cin » c;
         f = flaeche(a, b, c);
         v = volumen(a, b, c);}

protected:
    double flaeche(double x, double y, double z)
        {return(2 * (x * y + x * z + y * z));}
    double volumen(double x, double y, double z)
        {return(x * y * z);}

private:
    double a, b, c, x, y, z;
};

class Wuerfel : public Quader {   // abgel. konkr. Klasse

public:
    Wuerfel() {a = 0;}            // Constructor
    void lesen()
        {cout « "Laenge: "; cin » a;
         f = flaeche(a, a, a);
         v = volumen(a, a, a);}

protected:

private:
    double a;
};


// Hauptprogramm

int main()
{
int x = 0;
char figur[32];

    cout « "\nFlaechen- und Koerperberechnung\n\n";
    cout « "Welche Figur? ";
    cin » figur;
    cout « "\nFigur: " « figur « endl;

    // Stringvergleiche, erforderlich, weil in der
    // switch-Anweisung nur eine int-Variable stehen kann.

    if (!(strcmp(figur, "Kreis"))) x = 21;
    if (!(strcmp(figur, "Rechteck"))) x = 22;
    if (!(strcmp(figur, "Quadrat"))) x = 23;

    if (!(strcmp(figur, "Kugel"))) x = 31;
    if (!(strcmp(figur, "Quader"))) x = 32;
    if (!(strcmp(figur, "Wuerfel"))) x = 33;

    // Erzeugen des passenden Objektes f. Gilt wie
    // jede Deklaration nur innerhalb des Blockes {},
    // weshalb die Methoden lesen() und schreiben()
    // in jedem Block vorkommen muessen.

    switch (x)
    {
        case 21: {
            Kreis f;          // Erzeugen des Objektes f
```

```
            f.lesen();
            f.schreiben();
            break;
        }
        case 22: {
            Rechteck f;
            f.lesen();
            f.schreiben();
            break;
        }
        case 23: {
            Quadrat f;
            f.lesen();
            f.schreiben();
            break;
        }
        case 31: {
            Kugel f;
            f.lesen();
            f.schreiben();
            break;
        }
        case 32: {
            Quader f;
            f.lesen();
            f.schreiben();
            break;
        }
        case 33: {
            Wuerfel f;
            f.lesen();
            f.schreiben();
            break;
        }
        default:
            cout « "Keine gueltige Figur." « endl;
            exit(-1);
    }
return 0;
}
```

Quelle 1.64 : C++-Programm zur Berechnung geometrischer Formen

Geometrische Formen legen eine objektorientierte Programmierung nahe, da sie eine Hierarchie bilden, ähnlich wie Pflanzen oder Tiere. Im Beispiel wird als erstes eine abstrakte Basisklasse `Form` definiert, die das enthält, was allen Formen gemeinsam ist. Das ist nicht viel und steht in den beiden rein virtuellen Funktionen `lesen()` und `schreiben`.

Aus der Klasse `Form` werden die beiden immer noch abstrakten Klassen `Flaeche` und `Koerper` abgeleitet. Von Flächen läßt sich sagen, dass sie einen Umfang und einen Inhalt haben, ausgedrückt durch die beiden rein virtuellen Funktionen `umfang()` und `inhalt()`. Für Körper haben wir entsprechend die abstrakte Klasse `Koerper` mit den rein virtuellen Funktionen `flaeche()` (Oberfläche) und `volumen()`.

Im nächsten Schritt gelangen wir endlich zu konkreten Klassen. Aus der Klasse `flaeche` werden die Klassen `Kreis` und `Rechteck` abgeleitet, aus der Klasse `Rechteck` noch die Klasse `Quadrat`. Während aus abstrakten Klassen nur weitere, abstrakte oder konkrete Klassen abgeleitet werden können, lassen sich aus konkreten Klassen weitere konkrete Klassen ableiten oder Objekte bilden.

Bei den Körpern leiten wir analog aus der abstrakten Klasse `Koerper` die konkreten Klassen `Kugel` und `Quader` ab, aus `Quader` nochmals `Wuerfel`.

In den konkreten Klassen erhalten die virtuellen Funktionen auch einen konkreten Inhalt, das heißt, die Platzhalter werden mit den Formeln für den Flächeninhalt eines Kreises oder Rechtecks besetzt usw. Diese Formeln sehen für jede konkrete geometrische Form anders aus.

Bei einer Klassenableitung wie:

```
class Kreis : public Flaeche { }
```

bedeutet das Schlüsselwort `public`, dass von der zugrunde liegenden Klasse (Basisklasse) die public members als public und die protected members als protected geerbt werden. Die private members werden in keinem Fall vererbt. Hätten wir dagegen das Wort `private` gebraucht, so wären die vererbten members in der abgeleiteten Klasse privat geworden.

Das Erbrecht zwischen Klassen ist noch differenzierter, auch Freunde können erben, und eine Klasse kann aus zwei Basisklassen abgeleitet werden, aber erstmal muß Obiges verstanden und geübt werden. Die ausgefeilte Klassenhierarchie hat den Vorteil, dass man auf jeder Stufe genau das festlegt, was sich dort festlegen läßt, nicht mehr und nicht weniger. Kontrollen und Änderungen werden stets in einer bestimmten Stufe vorgenommen.

Das Hauptprogramm `main()` ist vergleichsweise trivial. Nach ein bißchen Benutzerdialog werden in einer `switch`-Anweisung die ausgewählten Objekte erzeugt und deren Methoden aufgerufen, nämlich die ursprünglich virtuellenund in den konkreten Klassen definierten Funktionen `lesen()` und `schreiben()`. Die wesentliche Arbeit steckt in den Klassen.

1.6.4 Memo Klassen

- Bei einem abstrakten Datentyp ist das Innenleben von der Außenseite streng getrennt (Black Box).
- Klassen sind in einer Programmiersprache formulierte Beschreibungen abstrakter Datentypen.
- Objekte sind Verwirklichungen von Klassen, so wie Variable Verwirklichungen von Typen sind.
- Eine Klasse oder ein Objekt enthält Daten (data members) und Methoden (member functions).
- Daten und Methoden sind öffentlich (public), geschützt (protected) oder privat. Die Daten sind meist privat. Mindestens eine Methode muß öffentlich sein, sonst nützt die Klasse nicht viel.
- Aus Klassen können Unterklassen abgeleitet werden, die die öffentlichen und geschützten (protected) Daten und Methoden erben. Die Klassen bilden Hierarchien.
- Von einer abstrakten Klasse können nur Unterklassen, aber keine Objekte abgeleitet werden. Meist in den oberen Etagen der Hierarchie anzutreffen.

1.6.5 Übung Klassen

Es gibt Aufgaben, deren Struktur eine Modellierung durch eine Klassenhierarchie nahelegt. Bei anderen hinwiederum wirkt die Objektorientierung verkrampft. Mit C/C++ sind alle Wege offen.

Überlegen Sie, welche Klassen und Objekte man bei der Aufgabe zur Weganalyse zweckmäßig einrichtet. Sind die Wegstrecken oder die Fahrzeuge/Personen als Objekte anzusehen? Skizzieren Sie – ohne genau auf die Syntax zu achten – eine Klassenhierarchie samt Daten und Methoden.

In dem Beispielprogramm zur Befeuerung von Binnenschiffen kann man sich gut eine Hierarchie von Fahrzeugen und entsprechenden Klassen vorstellen, wobei an der Spitze die Klasse der *Hohlkörper von nicht ganz unbedeutender Größe* steht.

Auch bei dem Beispiel des Vokabeltrainers ist eine Hierarchie denkbar. Wie könnten die Klassen, Daten und Methoden aussehen? Welche Vorteile hätte hier die Objektorientierung vor der prozeduralen Denkweise?

1.7 Klassen-Bibliotheken

1.7.1 C++-Standardbibliothek

Standardbibliotheken sind eine Ergänzung der Compiler, ohne die man nicht weit kommt. Die Benutzer betrachten sie als festen Bestandteil der Compiler, obwohl sie im strengen Sinn nicht Bestandteil der Sprache sind. Zu C++ gehört ebenso wie zu C eine Standardbibliothek, deren Umfang und Funktionalität durch eine ISO/ANSI-Norm festgelegt ist. Im wesentlichen gehören dazu:

- die C-Standardbibliothek (damit man sie nicht extra zu nennen braucht),
- Input/Output-Klassen wie `basic_ios`,
- String-Klassen wie `basic_string`,
- numerisches Klassen wie `complex`,
- Klassen zur Ausnahmebehandlung wie `exception`,
- sonstige Klassen wie `pair` und Klassen zur Lokalisation (Anpassung an örtliche Gegebenheiten),
- die Standard Template Library (STL)

Die C++-Standardbibliothek ist wesentlich umfangreicher als die C-Standardbibliothek, sie erfordert mehr Zeit zum Einarbeiten, aber sie spart viel Mühe. Es ist Zeitverschwendung, die Klasse *Rad* neu zu erfinden. Die Standard Template Library spielt eine besondere Rolle, weil sie einige neue Begriffe in C++ einbringt.

1.7.2 Standard Template Library (STL)

Ein wichtiger Schritt vorwärts in der Standardisierung von C++ war die Annahme der **Standard Template Library** (STL) als Erweiterung der C++-Standard-Bibliothek durch das ANSI-Komitee im Jahr 1993. Die STL enthält fünf Gruppen von Komponenten:

- Allgemeine Algorithmen,
- Iteratoren,
- Container,
- Funktionen,
- Adaptoren.

Wenn man in einem C-Programm ein Array linear (sequentiell) nach einem bestimmten Wert durchsuchen will, sieht die Funktion für Ganzzahlen anders aus als für Strings, obwohl der Suchalgorithmus derselbe ist. Die Algorithmen der STL sind dagegen allgemein gültig, indem sie mit Hilfe von **Templates** den Algorithmus vom Datentyp trennen. Ein Template (Vorlage, Muster, Schablone) ist eine allgemeine Vorstufe zu einer Funktion oder einer Klasse, der die Typen der verwendeten Daten als Parameter mitgegeben werden – ähnlich wie bei einem Funktionsaufruf die Typen der Argumente. Der Compiler erzeugt dann aus dem Template die gewünschte Klasse oder Funktion und nimmt so dem Programmierer Arbeit ab.

Iteratoren sind eine Verallgemeinerung der Pointer. Sie werden eingesetzt, um auf Elemente von Containern zuzgreifen, so wie man mittels Pointerarithmetik auf die Elemente eines Arrays zugreift. Im Gegensatz zu Pointern bringen sie jedoch eine gewisse eigene Intelligenz mit, so daß der Programmierer – wenn er erst einmal ihre Funktionsweise begriffen hat – weniger Arbeit aufzuwenden braucht.

Unter **Containern**, auch Collection genannt, werden Datenstrukturen (Klassen) verstanden, die andere Datenstrukturen oder Objekte enthalten. Damit lässt sich eine Gruppe von Objekten unter einem Namen gemeinsam handhaben. **Sequentielle Container** speichern ihre Objekte in einer Reihe (linear), auf sie kann entweder der Reihe nach oder wahlfrei zugegriffen werden. Daneben gibt es die **assoziativen Container**, deren Objekte über einen

Schlüssel oder Index zugänglich sind. Der einfachste sequentielle Container ist der Vektor, ein Array variabler Größe. Der einfachste assoziative Container ist die Menge (Set), in der jeder Schlüssel nur einmal vorkommen darf. Zu jeder Art von Containern gehört ein Satz von Methoden. Auch hier braucht sich der Programmierer nicht um die Einzelheiten der Speicherung und der Zugriffe zu kümmern, das hat die Bibliothek bereits erledigt.

Ein **Adaptor** schließlich macht das, was auch andere Adapter machen: er paßt das Aussehen einer Schnittstelle an neue Erfordernisse an, er verpackt einen Iterator oder einen Container in eine neue Umhüllung. Damit wird nicht ein neues Objekt geschaffen, sondern nur seine Außenseite verändert. Das ist oftmals effektiver, als ein neues Objekt zu schreiben.

Um die STL ganz zu verstehen, muß man sie benutzen. Da sie auf einer hohen Abstraktionsebene zu Hause ist, machen sich ihre Vorteile erst bei umfangreicheren Aufgaben bemerkbar. Für Hallo-Welt-Programme ist sie einige Nummern zu groß. Wir verweisen daher auf das Buch von ROBERT ROBSON und auf unsere Technik-Seite im WWW.

1.7.3 C-XSC

1.7.3.1 Was ist C-XSC?

Für numerische Aufgaben wurde im Institut für Angewandte Mathematik der Universität Karlsruhe eine **Klassenbibliothek C-XSC** als Ergänzung eines C++-Compilers entwickelt[33]. Das Kürzel XSC ist als *Extended Scientific Computing* zu deuten. Die wichtigsten Bestandteile von C-XSC sind:

- Arithmetik reeller und komplexer Zahlen sowie Intervallarithmetik mit mathematisch bestimmten Eigenschaften,
- dynamische Vektoren und Matrizen mit zur Laufzeit veränderbarer Größe,
- Teilfelder (Subarrays) aus Vektoren und Matrizen,
- arithmetische Operatoren und mathematische Standardfunktionen von hoher, bekannter Genauigkeit,
- dynamische Langzahlarithmetik mit zughörigen Standardfunktionen,
- Rundungskontrolle bei der Ein- und Ausgabe,
- Behandlung bestimmter Fehler (z. B. Überschreiten der Indexgrenzen),
- Bibliothek mit Routinen zur Lösung von Standardproblemen der numerischen Analysis.

Zusammen mit der Klassenbibliothek C-XSC geht C++ in der Behandlung numerischer Aufgaben über FORTRAN und andere Programmiersprachen hinaus.

1.7.3.2 Datentypen, Operatoren und Funktionen

C-XSC stellt folgende einfache numerische Datentypen zur Verfügung:

`real, interval, complex, cinterval` (=complex interval)

samt den zugehörigen arithmetischen und relationalen Operatoren und den mathematischen Standardfunktionen. Alle vordefinierten arithmetischen Operatoren liefern Ergebnisse mit einer Genauigkeit von wenigstens einer Einheit der letzten Stelle. Auf diese Weise sind sie maximal genau im Sinne des wissenschaftlichen Rechnens. Die von den arithmetischen Operatoren vorgenommenen Rundungen lassen sich durch den Gebrauch der Datentypen `interval` und `cinterval` steuern. Funktionen zur Typumwandlung sind für alle mathematisch sinnvollen Kombinationen verfügbar. Alle mathematischen Standardfunktionen für einfache numerische Datentypen können mit ihrem gewohnten Namen aufgerufen werden und liefern Ergebnisse von garantierter hoher Genauigkeit für beliebige Argumente aus dem

[33]Dieser Abschnitt ist die gekürzte Übersetzung eines Aufsatzes (1992) von Dipl.-Math. ANDREAS WIETHOFF, Mitarbeiter des genannten Institutes, siehe `www.uni-karlsruhe.de/~iam/html/language/cxsc/cxsc.html`.

Definitosnbereich. Die Standardfunktionen für die Datentypen `interval` und `cinterval` liefern scharfe Einschlüsse der Wertebereiche.

Zu den oben genannten einfachen Datentypen kommen die entsprechenden Felddatentypen (Vektoren und Matrizen):

```
rvector, ivector, cvector, civector,
rmatrix, imatrix, cmatrix, cimatrix
```

Der Anwender kann Speicherplatz für diese Arrays zur Laufzeit zuordnen und freigeben. Auf diese Weise kann dasselbe Programm für Arrays unterschiedlicher Größe benutzt werden, begrenzt nur durch den Arbeitsspeicher. Es wird nicht mehr Speicher von den Daten belegt, als wirklich benötigt wird. Beim Zugriff auf Arrays wird der Indexbereich zur Laufzeit geprüft, um Programmabstürze durch unzulässige Speicherzugriffe (memory faults) zu verhindern.

Hier ein Beispiel für die dynamische Größenänderung einer Matrix:

```
...
int n, m;
cout << "Dimensionen n, m eingeben: ";
cin  >> n >> m;

imatrix B, C, A(n,m);        /* A[1][1] ... A[n][m] */
Resize(B,m,n);               /* B[1][1] ... B[m][n] */
...
C = A * B;                   /* C[1][1] ... C[n][n] */
```

Mit Hilfe des C++-ostream-Objektes `cout` wird ein String nach `stdout` geschrieben, dann werden mit `cin` die beiden Indexgrenzen `n` und `m` von `stdin` eingelesen. Die Vereinbarung eines Vektors oder einer Matrix ohne Angabe der Indexgrenzen liefert einen Vektor mit einer Komponente oder eine Matrix mit je einer Zeile und Spalte. Speicher für diese Objekte wird erst an einer späteren Stelle im Programm zur Laufzeit zugewiesen. Die Matrix `A` wird gleich in der richtigen Größe angelegt. Die `Resize`-Anweisung bringt die Matrix `B` zur Laufzeit an dieser Programmstelle auf die erforderliche Größe.

Die Belegung von Speicherplatz für einen Vektor oder eine Matrix kann auch implizit – ohne eine ausdrückliche Anweisung wie `Resize` – durch eine Zuweisung erfolgen. Falls die Indexgrenzen des Objektes auf der rechten Seite einer Zuweisung nicht mit den Indexgrenzen des Objektes auf der linken Seite übereinstimmen, wird das linke Objekt angepaßt, wie es hier mit der Matrix `C` geschieht.

Der dynamisch zugewiesene Speicherplatz eines lokal vereinbarten Arrays wird automatisch beim Verlassen des Gültigkeitsbereiches freigegeben. Hinsichtlich Lebensdauer und Gültigkeitsbereich besteht kein Unterschied zu den aus C gewohnten Arrays.

Die Größe eines Vektors oder einer Matrix kann jederzeit durch Aufrufen der Funktionen `Lb()` und `Ub()` für die untere bzw. obere Indexgrenze ermittelt werden.

1.7.3.3 Teilfelder von Vektoren und Matrizen

C-XSC stellt eine eigene Schreibweise für die Handhabung von **Teilfeldern** (Subarrays) von Vektoren und Matrizen zur Verfügung. Subarrays sind beliebige rechteckige Ausschnitte aus Arrays. Alle vordefinierten Operatoren nehmen auch Subarrays als Operanden an. Ein Subarray einer Matrix oder eines Vektors wird über den ()-Operator oder über den []-Operator angesprochen. Der ()-Operator bezeichnet ein Subarray eines Objektes vom selben Typ wie das ursprüngliche Objekt. Ist beispielsweise A eine reelle n x n-Matrix, dann ist A(i,i) die linke obere i x i-Submatrix. Man beachte, daß die Klammern in der Deklaration eines dynamischen Vektors oder einer ebensolchen Matrix kein Subarray bezeichnen, sondern den Indexbereich des anzulegenden Objektes einschließen. Der []-Operator erzeugt ein Subarray eines niedrigeren Typs. Wenn A eine Matrix vom Typ n x n-`rmatrix` ist, dann ist A[i] die i-te Zeile von A mit dem Typ `rvector`, und A[i][j] ist das (i,j)-te Element von A mit dem Typ `real`.

Beide Arten des Zugriffs auf Subarrays können auch verbunden werden, beispielsweise ist A[k](i,j) ein Subvektor von Index i bis Index j des k-ten Zeilenvektors der Matrix A.

Den Gebrauch von Subarrays zeigt das folgenden Beispiel, das die LU-Faktorisierung einer n * n-Matrix A beschreibt:

```
for (j = 1; j <= n - 1; j++) {
  for (k = j + 1; k <= n; k++) {
    A[k][j]        = A[k][j] / a[j][j];
    A[k](j + 1, n) = A[k](j + 1, n) - A[k][j] * A[j](j + 1, n);
  }
}
```

Dieses Beispiel nutzt zwei wichtige Möglichkeiten von C-XSC. Erstens sparen wir uns eine Schleife durch die Verwendung von Subarrays. Das vereinfacht das Programm. Zweitens ist der obige Programmausschnitt unabhängig vom Typ der Matrix A (`rmatrix, imatrix, cmatrix` oder `cimatrix`), da alle arithmetischen Operatoren so vordefiniert sind wie man es in der Mathematik erwartet.

1.7.3.4 Genaue Auswertung von Ausdrücken

Beim Auswerten arithmetischer Ausdrücke spielt die Genauigkeit eine entscheidende Rolle in vielen Algorithmen. Auch wenn alle arithmetischen Operatoren und Standardfunktionen für sich allein maximaler genau sind, so liefern zusammengesetzte Ausdrücke doch nicht notwendigerweise Ergebnisse maximaler Genauigkeit. Deshalb sind Verfahren entwickelt worden, die numerische Ausdrücke mit hoher und auf mathematischem Wege garantierter Genauigkeit auswerten.

Eine besondere Art solcher Ausdrücke sind die sogenannten **Skalarproduktausdrücke**. Sie sind als eine Summe einfacher Ausdrücke definiert. Ein einfacher Ausdruck ist entweder eine Variable, eine Konstante oder ein einzelnes Produkt von zweien solcher Objekte. Die Variablen dürfen vom Typ Skalar, Vektor oder Matrix sein. Nur die mathematisch sinnvollen Operationen sind für die Addition und die Multiplikation zugelassen. Das Ergebnis der Auswertung eines solchen Ausdrucks ist entweder ein Skalar, ein Vektor oder eine Matrix. In der numerischen Analysis sind Skalarprodukte von entscheidender Bedeutung. Beispielsweise gründen sich Verfahren zur Fehlerkorrektur oder zur iterativen Verbesserung bei linearen oder nichtlinearen Aufgaben auf Skalarprodukte. Eine Auswertung dieser Ausdrücke mit maximaler Genauigkeit vermeidet Fehler durch Auslöschung. Für eine Auswertung mit einer Genauigkeit von einer Einheit der letzten Stelle stellt C-XSC die folgenden **Dotprecision-Datentypen** zur Verfügung:

```
dotprecision, cdotprecision, idotprecision, cidotprecision
```

Zwischenergebnisse eines Skalarproduktes können ohne jeden Rundungsfehler in einer Dotprecision-Variablen errechnet und gespeichert werden. Die folgende Funktion berechnet eine maximal genaue Einschließung des Defektes b - Ax eines linearen Gleichungssystems Ax = b :

```
ivector defect (rvector b, rmatrix A, rvector x)
{
    idotprecision accu;
    ivector INCL (Lb(x), Ub(x));

    for (int i = Lb(x); i <= Ub(x); i++) {
        accu = b[i];
        accumulate(accu, -A[i], x);
        INCL[i] = rnd(accu);
    }
    return INCL;
}
```

Quelle 1.65 : C-XSC-Funktion `defect()` zur Defekteinschließung

In obigem Beispiel berechnet die Funktion `accumulate()` die Summe:

$$\sum_{j=1}^{n} -A_{ij} \cdot x_j$$

und addiert das Ergebnis zu dem Dotprecision-Akkumulator `accu` ohne Rundungsfehler. Die `idotprecision`-Variable `accu` wird mit `b[i]` initialisiert. Schließlich wird der Wert im Akkumulator maximal genau auf das Standard-Intervall `INCL[i]` gerundet. Auf diese Weise sind die Grenzen von `INCL[i]` entweder gleich oder zwei beanchbarte Gleitkommazahlen.

Für alle Dotprecision-Datentypen steht ein verringerter Satz vordefinierter Operatoren zur Verfügung, um fehlerfreie Ergebnisse zu berechnen. Die überladene Skalarprodukt-Funktion `accumulate()` und die Rundungsfunktion `rnd()` sind für alle sinnvollen Typkombinationen verfügbar.

1.7.3.5 Dynamische Langzahl-Arithmetik

Neben den Klassen `real` und `interval` gibt es die dynamischen Klassen `l_real` (long real) und `l_interval` (long interval) ebenso wie die entsprechenden dynamischen Vektoren und Matrizen samt allen arithmetischen und relationalen Operatoren und allen Standardfunktionen mit mehrfacher Genauigkeit. Die Rechengenauigkeit läßt sich vom Benutzer während der Laufzeit kontrollieren. Mittels Ersetzen der Typen `real` und `interval` durch `l_real` und `l_interval` in den Deklarationen wird ein Anwendungsprogramm zu einem Programm mit mehrfacher Genauigkeit. Dieses Konzept gibt dem Benutzer ein mächtiges und einfach zu handhabendes Werkzeug zur Fehleranalyse in die Hand. Weiterhin ist es möglich Programme zu schreiben, die numerische Ergebnisse mit einer vom Benutzer vorgegebenen Genauigkeit liefern, indem man intern die Rechengenauigkeit zur Laufzeit in Abhängigkeit von den Fehlerschranken der Zwischenergebnisse anpaßt.

Alle vordefinierten Operatoren für die Typen `real` und `interval` sind auch für die Typen `l_real` und `l_interval` verfügbar. Zusätzlich sind auch alle möglichen Kombinationen von Operatoren für Typen einfacher und mehrfacher Genauigkeit vorhanden. Im folgenden wird ein Programm mit einfacher Genauigkeit und sie entsprechende Version mit mehrfacher Genauigkeit gezeigt:

```
main()
{
    interval a, b;              /* Standard-Intervall */
    a = 1.0;                    /* a = [1.0, 1.0] */
    b = 3.0;                    /* b = [3.0, 3.0] */
    cout << "a/b = " << a/b;    /* a/b = [0.333333333333,
                                       0.333333333334] */
}
```

Quelle 1.66 : C-XSC-Programm einfacher Genauigkeit

```
main()
{
    l_interval a, b;            /* Langzahl-Intervall */
    a = 1.0;
    b = 3.0;
    stagprec = 2;               /* Globale int-Variable */
    cout << "a/b = " << a/b;    /* a/b =
                              [0.333333333333333333333333,
                              0.333333333333333333333334] */
}
```

Quelle 1.67 : C-XSC-Programm mehrfacher Genauigkeit

Zur Laufzeit bestimmt die vordefinierte globale int-Variable `stagprec` (staggered precision) die Rechengenauigkeit der Langzahl-Arithmetik in Schritten einer `real`-Zahl (64-Bit-Maschinenwort). Die Genauigkeit einer Langzahl ist als die Anzahl von `real`-Zahlen definiert, die zur Speicherung der langen Zahl verwendet werden. Ein Objekt des Typs `l_real` oder `l_interval` kann seine Genauigkeit zur Laufzeit ändern. Komponenten eines Vektors oder einer Matrix dürfen von unterschiedlicher Genauigkeit sein. Alle Standardfunktionen und sonstigen Funktionen der Langzahl-Arithmetik berechnen numerische Ergebnisse mit einer Genauigkeit, die durch den augenblicklichen Wert von `stagprec` bestimmt ist. Speicherzuweisung, das Ändern der Größe von Arrays und das Arbeiten mit Subarrays verlaufen ähnlich wie bei den entsprechenden Datentypen einfacher Genauigkeit.

1.7.3.6 Ein- und Ausgabe in C-XSC

Unter Verwendung des Stream-Konzeptes und der überladbaren Operatoren << und >> von C++ ermöglicht C-XSC das Runden und Formatieren aller seiner Datentypen während der Ein- und Ausgabe, auch für die Dotprecision- und Langzahldatentypen. Ein-/Ausgabe-Parameter wie die Richtung der Rundung, Feldbreite usw. benutzen auch die überladenen I/O-Operatoren zum Formatieren der Ein- und Ausgabe. Falls ein neuer Satz von I/O-Parametern verwendet werden soll, kann der alte auf einem internen Stack gespeichert und bei Bedarf zurückgeholt werden. Das folgende Beispiel zeigt die Möglichkeiten von C-XSC zur Ein- und Ausgabe:

```
main()
{
    real a, b; interval c;

    cout << "Bitte reelle Zahlen a, b eingeben: ";
    cout << RndDown;
    cin  >> a;                     /* lies a abwaerts gerundet */
    cout << RndUp;
    cin  >> b;                     /* lies b aufwaerts gerundet */
    "[0.11, 0.22]" >> c;           /* String nach Intervall */
    cout << SaveOpt;               /* I/O-Parameter auf Stack */
    cout << SetPrecision(20, 16);  /* Feldbreite, Stellen */
    cout << Hex;                   /* hexadezimale Ausgabe */
    cout << c  << RestoreOpt;      /* alte I/O-Param. zurueck */
}
```

Quelle 1.68 : C-XSC-Programm mit formatierter Ein- und Ausgabe

1.7.3.7 C-XSC-Numerikbibliothek

Die C-XSC-Numerikbibliothek ist eine Sammlung von Funktionen und Programmen zur Lösung von Standardaufgaben der numerischen Analysis mit garantierter Genauigkeit der Ergebnisse. Folgende Bereiche werden abgedeckt:

- Auswertung und Nullstellen von Polynomen,
- Lineare Systeme, Matrizeninvertierung,
- Eigenwerte, Eigenvektoren,
- Schnelle Fourier-Transformation,
- Nullstellen nichtlinearer Gleichungen,
- Anfangswertprobleme bei gewöhnlichen Differentialgleichungen.

1.7.3.8 Beispiel Intervall-Newton-Verfahren

Zu bestimmen sei der Einschluß einer Nullstelle einer reellen Funktion $f(x)$. Die erste Ableitung $f'(x)$ sei stetig im Intervall $[a, b]$ und es gelte:

$$0 \notin \{f'(x),\, x \in [a, b]\} \quad \textbf{und} \quad f(a) \cdot f(b) < 0.$$

Falls X_n eine Einschließung der Nullstelle ist, dann wird eine verbesserte Einschließung X_{n+1} mittels der Formel:

$$X_{n+1} := \left(m(X_n) - \frac{f(m(X_n))}{f'(X_n)} \right) \cap X_n$$

berechnet, wobei $m(X)$ ein Punkt im Intervall X ist, üblicherweise die Mitte. Die Funktion sei in diesem Beispiel:

$$f(x) = \sqrt{x} + (x + 1) \cdot \cos x$$

Nun das Programm:

```
/* C-XSC-Programm Newton-Verfahren mit Intervallen
   Funktion f(x) = sqrt(x) + (x + 1) cos(x)
   Inst. Angewandte Mathematik, Universitaet Karlsruhe */

#include "interval.hpp"      // Interval arithmetic package
#include "imath.hpp"         // Interval standard functions

interval f(real& x)          // Function f()
{
interval y;
y = x;                       // Use interval arithmetic
return (sqrt(y) + (y + 1.0) * cos(y));
}

interval deriv(interval& x) // Derived function f'()
{
return (1.0 / (2.0 * sqrt(x)) + cos(x) - (x + 1.0) * sin(x));
}

int criter(interval& x)      // Function testing f(a)*f(b) < 0
{
interval Fa, Fb;
Fa = f(Inf(x));
Fb = f(Sup(x));
return (Sup(Fa * Fb) < 0.0 && !(0.0 <= deriv(x)));
                             // <= means element of
}

main()
{
interval y, y_old;

cout << "Please enter starting interval: "; cin >> y;
cout << "SetPrecision(20, 12);
if (criter(y))
    do {
        y_old = y;
        cout << "y = " << y << endl;
        y = (mid(y) - f(mid(y)) / deriv(y)) & y;
                              // & means intersection
```

```
    } while (y != y_old);
else
    cout << "Criterion not satisfied!" << endl;
}
```

Quelle 1.69 : C-XSC-Programm Intervall-Newton-Verfahren

Weitere Beispiele finden sich in dem Buch von RUDI KLATTE und anderen, siehe Anhang H *Zum Weiterlesen* auf Seite 234.

1.7.4 X11-Programmierung mit dem Qt-Toolkit

Der Quasar-Toolkit (Qt) ist eine **Widget-Bibliothek** der Firma Troll Tech (`www.troll.no`), die sowohl für das X Window System (X11) als auch für Microsoft Windows erhältlich ist. Für die Entwicklung freier UNIX-Software ist die Nutzung der Bibliothek kostenlos; vor einiger Zeit hat Troll Tech die Bibliothek unter eine OpenSource-Lizenz (`www.opensource.org`) gestellt, so daß auch Änderungen an den Quellen erlaubt sind. Bekannt wurde Qt als Basis der Arbeitsumgebung KDE (`www.kde.org`).

Die X11-Programmierung gestaltet sich mit Qt deutlich einfacher als mit anderen Bibliotheken (Motif) und ist geprägt einerseits durch die Verwendung von C++, andererseits durch den Mechanismus von **Signalen** und **Slots**; beides trägt wesentlich zur effizienteren und weniger fehleranfälligen Programmierung bei.

Die libQt eignet sich auch hervorragend, um das Konzept der objektorientierten Programmierung besser zu verstehen: Benötigt man beispielsweise eine neue Art von Knopf, entwickelt man auf der Basis der abstrakten Klasse `QButton` eine neue Klasse, die damit alle Grundeigenschaften von Knöpfen erbt (abstrakt bedeutet hier, daß die Klasse einige Funktionen enthält, die implementiert werden müssen, bevor die Klasse verwendet werden kann); neu schreiben muß man nur noch die Funktionen `drawButton()` und `drawButtonLabel()`, die für die eigentliche Darstellung des Knopfes verantwortlich zeichnen. Diese Vorgehensweise läßt sich verallgemeinern: Um ein neues Widget zu entwickeln, geht man von einem in der Bibliothek bereits vorhandenen aus, das in seinen Eigenschaften dem gewünschten Ergebnis möglichst nahe kommt, und schreibt nur diejenigen Funktionen neu, die sich unterscheiden beziehungsweise dazukommen.

Signale und Slots gestalten die Kommunikation zwischen verschiedenen Programmteilen. Ein Widget (z. B. ein Knopf) sendet bei einem bestimmten Ereignis (der Knopf wird vom Benutzer betätigt) ein Signal aus (in diesem Fall das Signal `clicked()`), das vorher mit einem Slot in einem anderen Programmteil verbunden wurde. Viele Widgets beinhalten dearartige Slots; selbstverständlich kann man auch in seinen eigenen Klassen Funktionen als Slots deklarieren.

Um eigene Klassen mit Slots zu implementieren, ist die Verwendung des **Meta Object Compilers** `moc` vonnöten, der, auf die Klassendeklaration angewandt, einige Makros ersetzt; dieser Schritt ist notwendig, da das Konzept von Signalen und Slots nicht in C++ enthalten ist. Üblicherweise wird man die Klassendeklarationen in einer Include-Datei unterbringen, auf das dann der Meta Object Compiler angewandt wird. Die Ausgabe von `moc` leitet man in eine Datei mit der Kennung `.moc` um, die anstelle der Include-Datei mit den Klassendeklarationen eingebunden wird.

Das folgende Beispiel besteht aus einem Makefile, einer Include-Datei und dem eigentlichen Programm. Das Programm öffnet ein Fenster und bringt in dessen Mitte eine Beschriftung in Form eines für derartige Zwecke klassischen Textes sowie an der Fensterunterseite einen Knopf an. Wird der Knopf betätigt, ändert sich die Beschriftung, und nach 1500 ms beendet sich das Programm.

```
# Makefile fuer qhello

# es werden die Include-Files von X11 und Qt benoetigt
INC = -I/usr/X11R6/include -I/usr/lib/qt/include
```

```
# gelinkt wird gegen die libX11 (in /usr/X11R6/lib)
# und die libqt (in /usr/lib, dort automatisch gesucht)
LIB = -L/usr/X11R6/lib -lX11 -lqt

# der verwendete C++-Compiler - hier GNU C++
CPP = g++

# benoetigt wird neben dem Source-Code das moc-File
all:    qhello.cpp qhello.moc
        $(CPP) -o qhello qhello.cpp $(INC) $(LIB)

# moc-File wird aus qhello.h gewonnnen;
# moc muss im PATH stehen
qhello.moc:     qhello.h
                moc qhello.h > qhello.moc
```

Quelle 1.70 : Makefile zu qhello.cpp

```
/* Include-File fuer qhello */

// Die Klasse HelloWidget erbt ihre Eigenschaften
// von der Klasse QWidget

class HelloWidget
 : public QWidget
{
   // Q_OBJECT; muss in jeder Klasse, die Signals oder
   // Slots implementiert, stehen
   Q_OBJECT;

   // die Konstruktor-Funktion (ohne void)
   public:
      HelloWidget(QWidget *parent = 0, const char *name = 0);

   // ein Signal, das diese Klasse aussendet
   signals:
      void neuerText( const char * );

   // Pointer fuer verwendete Widgets
   private:
      QLabel *helloLabel;
      QPushButton *quitButton;
      QTimer *quitTimer;

   // ein Slot, der spaeter mit quitButton verbunden wird
   private slots:
      void tschuess();
};
```

Quelle 1.71 : Include-Datei zu qhello.cpp

```
/* qhello.cpp - Beispiel fuer die Programmierung mit Qt */

#include <qapplication.h>  // benoetigt jedes Qt-Programm
#include <qwidget.h>       // fuer unser Hauptfenster
#include <qlabel.h>        // fuer die Beschriftung
#include <qpushbutton.h>   // fuer den Quit-Button
#include <qtimer.h>        // fuer zeitverzoegertes Beenden

// an dieser Stelle wird das moc-File eingebunden;
// es enthaelt die Prototypen fuer unsere Klasse
#include "qhello.moc"
```

```
// die Konstruktorfunktion unserer Klasse
HelloWidget::HelloWidget(QWidget *parent, const char *name)
 : QWidget( parent, name )
{
   // die Groesse des Hauptfensters setzen
   resize( 200, 100 );

   // eine Beschriftung wird erzeugt und plaziert
   helloLabel = new QLabel( "hello, world!", this );
   helloLabel->move( 60, 30 );

   // Signal neuerText()
   // dieses Objekts wird mit helloLabel verbunden
   // setText() ist Slot zum Setzen des Beschriftungstextes
   connect(this, SIGNAL(neuerText(const char *)), helloLabel,
      SLOT( setText( const char * ) ) );

   // quitButton wird erzeugt und plaziert...
   quitButton = new QPushButton( "Quit", this );
   quitButton->move( 0, 80 );
   quitButton->resize( 200, 20 );

   // ... und verbunden mit unserem Slot tschuess()
   connect(quitButton, SIGNAL(clicked()), this, \
                             SLOT(tschuess()));
}

// unser Slot tschuess(), der mit quitButton verbunden ist
void HelloWidget::tschuess()
{
   // Signal neuerText() aussenden (an helloLabel)
   emit( neuerText( "Tschuess!!!" ) );

   // einen Timer einrichten und starten, Laufzeit 1500 ms
   quitTimer = new QTimer( this );
   quitTimer->start( 1500 );

   // bei Ablauf des Timers wird Slot quit() der Haupt-
   // applikation ausgefuehrt, der das Programm beendet
   connect(quitTimer, SIGNAL(timeout()), qApp, \
                             SLOT(quit()));
}

// Hauptprogramm
int main( int argc, char **argv )
{
// QApplication ist die Klasse fuer die Hauptapplikation
// argc und argv muessen uebergeben werden, um z. B.
// geometry-Informationen auszuwerten
QApplication MeineAnwendung( argc, argv );

// unser Hauptwidget
HelloWidget MeinWidget;

// setzen als MainWidget der Applikation
MeineAnwendung.setMainWidget( &MeinWidget );

// und darstellen
MeinWidget.show();

// hiermit wird die Hauptapplikation ausgefuehrt
return MeineAnwendung.exec();
}
```

Quelle 1.72 : C++-Programm qhello.cpp mit Verwendung des Qt-Toolkit

Für das Hauptfenster werden ein neues Widget von der Widget-Oberklasse QWidget abgeleitet, das Beschriftung und Knopf erzeugt und das Signal clicked(), welches der Knopf bei Betätigung aussendet, mit dem klasseneigenen Slot tschuess() verbunden. Dieser Slot sendet das ebenfalls klasseneigene Signal neuerText() aus, das vorher mit dem Slot setText() des Beschriftungsfeldes verbunden wurde. Dieser Slot setzt, seinem Namen gemäß, den Text des Beschriftungsfeldes neu. Gleichzeitig wird ein Timer erzeugt und gestartet, der nach 1500 ms das Signal timeout() aussendet, welches wiederum mit dem Slot quit() der Hauptapplikation verbunden wird. Die Funktion dieses Slots ergibt sich ebenfalls aus seinem Namen.

Dieses Basisprogramm läßt sich unter Zuhilfename der exzellenten Online-Dokumentation zu Qt, die sich auf dem Webserver von Troll Tech (www.troll.no) findet, schnell erweitern. Probieren Sie es aus: Die Programmierung mit Qt ist auch für C++-Einsteiger und Anfänger in der X11-Programmierung einfach zu erlernen und zeitigt schnell Erfolgserlebnisse.

1.8 Überladen von Operatoren

Die arithmetischen Operationen auf Ganzzahlen unterscheiden sich von denen auf Gleitkommazahlen, insbesondere die Division. Dennoch verwenden wir dieselben Operatoren. Das Programm erkennt aus dem Zusammenhang, welche Art von Operation gefragt ist. Diese Möglichkeit, einem Operator je nach Zusammenhang verschiedene Bedeutungen (Definitionen) zu geben, ist in C++ verallgemeinert worden und läßt sich auf fast jeden Operator und jede Funktion ausdehnen.

Wir sehen uns am Beispiel eines C++-Programmes zur Berechnung von Primzahlen nach dem Divisionsverfahren an, wie ein Operator überladen wird:

```
/* prim.C, C++-Programm zur Berechnung von Primzahlen
   zu compilieren mit CC -o prim prim.C */

/* $Header: /home/debian/prog/quellen/primplus.tex,v 1.1.1.1 2005/02/06 19:44:00 wu

#include <iostream.h>
#include <stdio.h>

/***********/
/* Klassen */
/***********/

class PRIM {                  // Definition der Klasse PRIM

public:                       // nach aussen sichtbar,
                              // Methoden
        PRIM(int, int);       // Default Constructor
        PRIM& operator++();   // Ueberladen von ++ (Praefix)
        int get_count();      // Zugriff auf Anzahlen
        int get_prim();       // Zugriff auf Primzahlen

private:                      // nicht nach aussen sichtbar,
                              // Daten
        int prim_count;       // Anzahl der Primzahlen
        int prim_number;      // aktuelle Primzahl

};

/************/
/* Methoden */
/************/

PRIM::PRIM(int anzahl = 1, int primzahl = 2) // Constructor
{
        prim_count = anzahl;
```

```
        prim_number = primzahl;
}

// Folgende (triviale) Zugriffsmethoden sind erforderlich,
// da die Variablen prim_count und prim_number privat sind:

int PRIM::get_count()
{
        return(prim_count);
}

int PRIM::get_prim()
{
        return(prim_number);
}

/**************************************/
/* Ueberladen des ++ Praefix-Operators */
/**************************************/

PRIM& PRIM::operator++()
{
    if (prim_number == 2) {   // Test auf 2
        prim_number++;    // 2 + 1 = 3, naechste Primzahl
        ++prim_count;     // Inkrementierung der Anzahl
        return(*this);    // aktuelles Objekt zurueckgeben
    }

    for ( ; ; ) {         // ewige Schleife, bis break
        prim_number += 2; // naechste ungerade Zahl
        int prim_flag = 1;   // true
        int haelfte = (prim_number / 2);
        for (int i = 3; i < haelfte; i += 2) {
            if (((prim_number / i) * i) == prim_number) {
                prim_flag = 0;   // false, teilbar
                break;
            }
        }
        if (prim_flag) break;    // Verlassen der e. S.
    }
    ++prim_count;         // Inkrementierung der Anzahl
    return(*this);        // aktuelles Objekt zurueckgeben
}

/*****************/
/* Hauptprogramm */
/*****************/

int main(int argc, char *argv[])
{

int max;                  // Obergrenze
PRIM pzahl;               // Erzeugung des Objektes pzahl

// Obergrenze ermitteln

if (argc > 1) {
        sscanf(*(argv + 1), "%d", &max);
}
else {
        cout « "Obergrenze eingeben: ";
        cin » max;
}

// Ueberschrift ausgeben
```

```
cout « "\nPrimzahlen bis " « max « " inkl.: \n\n";

// Primzahlen berechnen und ausgeben

while (pzahl.get_prim() <= max) {
        cout « pzahl.get_prim() « '\n';
        // naechste Primzahl mittels ++-Operator:
        ++pzahl;
}

// Schlussbemerkung

cout « "Gesamtzahl: " « pzahl.get_count() - 1 « "\n\n";
return 0;
}
```

Quelle 1.73 : C++-Programm zur Berechnung von Primzahlen

Im Hauptprogramm wird zunächst die Obergrenze aus der Kommandozeile oder im Dialog ermittelt. Dann wird eine Überschrift ausgegeben. Der Witz kommt im Rumpf der `while`-Schleife. Offenbar wird die jeweils nächste Primzahl durch den Präfix-Operator `++` erreicht, mit dem man vor der Überladung nur Ganzzahlen um jeweils 1 inkrementieren konnte. Hinter dem überladenen Operator versteckt sich der Algorithmus zur Berechnung der nächsten Primzahl. Hier wird die Teilbarkeit der aktuellen Zahl `prime_number` dadurch ermittelt, daß sie durch die kleineren ungeraden Zahlen `i` ganzzahlig dividiert und gleich wieder mit dem Divisor multipliziert wird. Ergibt sich der alte Dividend, so verlief die Division ohne Rest, die aktuelle Zahl war teilbar und somit keine Primzahl. So geht es auch, wir werden im Programm 1.89 auf Seite 153 den modulo-Operator verwenden.

Die Überladung bewirkt, daß der Präfix-Operator `++`, angewandt auf ein Objekt der Klasse `PRIM`, die jeweils nächste Primzahl erzeugt. Man hätte auch einen anderen Operator nehmen können. Die für Klassen definierten Operatoren wie `::` dürfen nicht überladen werden, da sie gebraucht werden. Bei Operatoren wie der Bedingten Bewertung `?:` ist schwer vorstellbar, welche neue Bedeutung sie erhalten sollten.

1.9 Präprozessor

Beim Aufruf des Compilers wird der Quelltext als erstes von einem **Präprozessor** bearbeitet. Dieser führt die `define`- und `include`-Anweisungen aus und entfernt Kommentare sowie Zeichenpaare Backslash-Zeilenwechsel (verbindet Fortsetzungszeilen, der Zeilenwechsel muss unmittelbar auf den Backslash folgen).

1.9.1 define-Anweisungen

Die `define`-Anweisung dient zwei Zwecken: Sie definiert **symbolische Konstanten** sowie **Makros**. Eine symbolische Konstante ist ein konstanter Operand (auch ein String), der mit der `define`-Anweisung Namen und Wert erhält und im weiteren Programm nur noch mit seinem Namen aufgerufen wird:

```
#define MWST 1.15
....
brutto = netto * MWST;
```

Damit erleichtert man Änderungen, die sich so auf eine Stelle beschränken, und vermeidet das Auftauchen rätselhafter Zahlen (magic numbers) mitten im Programm. Bei Strings sind die Gänsefüßchen mit in die Definition zu nehmen und beim Aufruf nicht zu wiederholen.

Ein **Makro** ist eine nicht zu lange Folge von Ausdrücken und Anweisungen, alternativ zu einer kurzen Funktion. Das Makro wird zu in-line-Code und damit etwas schneller als die Funktion, die Parameterübergabe entfällt. Andererseits wird der ausführbare Code etwas länger. Ein Beispiel:

```
#define NEWFILE fprintf(stderr, "Datei?\n");
                    if (gets(name) == NULL) done()
....
if (argc < 2) NEWFILE;
```

Der Präprozessor ersetzt jedes `NEWFILE` im Programm buchstäblich durch die definierte Folge. Das spart Tipperei und verbessert die Übersichtlichkeit. Zeichenfolgen in Stringkonstanten (in Gänsefüßchen) werden nicht ersetzt; das Programmchen:

```
#define PI 3.14159

int main()
{
printf("Die Zahl PI ist %f\n", PI);
}
```

schreibt wie erhofft auf den Bildschirm:

```
Die Zahl PI ist 3.141590
```

Bei arithmetischen Makros muß man aufpassen, damit sie nicht infolge von Vorrangregeln in unbeabsichtigter Weise ausgeführt werden; am besten setzt man deutliche Klammern:

```
#define DELTA(a, b) ((a) - (b))
```

Hier können die Ausdrücke `a` und `b` aussehen, wie sie wollen, und in eine beliebige Umgebung eingebettet sein, das Makro wird immer als Differenz der beiden Ausdrücke aufgefaßt. Im Gegensatz dazu würde eine Definition ohne Klammern wie:

```
PROD(a, b) a * b
```

in einem Zusammenhang wie:

```
x = n * PROD(r - s, t + u);
```

zu folgender Ersetzung führen:

```
x = n * r - s * t + u;
```

wie man durch manuelles Einsetzen leicht nachvollzieht, und das ist vermutlich nicht das Gewünschte.

Mittels der `undef`-Anweisung widerruft man eine vorhergehende `define`-Anweisung. Das kommt selten vor. Hat man ein Makro und eine Funktion desselben Namens und will unbedingt die Funktion haben, so undefiniert man das Makro.

1.9.2 include-Anweisungen

Die `include`-Anweisung führt dazu, dass der Präprozessor die anschließend genannte Datei (Include-Datei, Header-Datei, Definitionsdatei) mit zu dem Programmtext lädt, bevor dieser zum eigentlichen Compiler gelangt. Die Include-Dateien ihrerseits enthalten symbolische Konstanten und Makros. Grundsätzlich jedoch dürfen sie alles enthalten, was nicht gegen die Regeln von C/C++ verstößt, also auch Kommentar oder Programmteile.

Die Namen der Standard-Include-Dateien sind in <> einzuschließen, die Namen eigener Include-Dateien in Gänsefüßchen. Letztere werden zuerst in dem Verzeichnis gesucht, in dem auch das Quellprogramm steht. Falls der Präprozessor dort nicht fündig wird, oder falls der Name in spitzen Klammern eingeschlossen ist, wird in vorgegebenen Verzeichnissen wie `/usr/include` gesucht. Durch Compileroptionen (siehe `man cpp`) läßt sich die Suche steuern.

Die Include-Dateien sind lesbar und finden sich im Verzeichnis `/usr/include`, vom Benutzer erstellte Include-Dateien im selben Verzeichnis wie die Programmquelle. Die Datei-Kennung `.h` ist üblich, aber nicht notwendig. Als Beispiel sehen wir uns die häufig gebrauchte Datei `<stdio.h>` in gekürzter Form an:

```
/* @(#) $Revision: 1.1.1.1 $ */

#ifndef _NFILE
#define _NFILE 60

#define BUFSIZ 1024

/*
 * buffer size for multi-character output to
 * unbuffered files
 */
#define _SBFSIZ 8

typedef struct {
int _cnt;
unsigned char *_ptr;
unsigned char *_base;
short _flag;
char _file;
} FILE;

/*
 * _IOLBF means that a file's output will be buffered
 * line by line.
 * In addition to being flags, _IONBF, _IOLBF and _IOFBF
 * are possible values for "type" in setvbuf.
 */
#define _IOFBF 0000
#define _IOREAD 0001
#define _IOWRT 0002
#define _IONBF 0004
#define _IOMYBUF 0010
#define _IOEOF 0020
#define _IOERR 0040
#define _IOLBF 0200
#define _IORW 0400

#ifndef NULL
#define NULL 0
#endif
#ifndef EOF
#define EOF (-1)
#endif

#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])

#ifndef lint
#define get(p)  ->_cnt < 0 ? _filbuf(p) : (int) *(p)->_ptr++)
#define putc(x, p)   (--(p)->_cnt < 0 ? \
     _flsbuf((unsigned char) (x), (p)) : \
     (int) (*(p)->_ptr++ = (unsigned char) (x)))
#define getchar()    getc(stdin)
#define putchar(x)   putc((x), stdout)
#define clearerr(p)  ((void) ((p)->_flag &= \
                     ~(_IOERR | _IOEOF)))
```

```
#define feof(p)       ((p)->_flag & _IOEOF)
#define ferror(p)     ((p)->_flag & _IOERR)
#define fileno(p)     (p)->_file
#else
void clearerr();
#endif not lint

extern FILE  _iob[_NFILE];
extern FILE  *fopen(), *fdopen(), *freopen(), *popen();
extern FILE  *tmpfile();
extern long  ftell();
extern void  rewind(), setbuf();
extern char  *ctermid(), *cuserid(), *fgets(), *gets();
extern char  *tmpnam();
extern unsigned char *_bufendtab[];

#endif NFILE
```

Quelle 1.74 : Include-Datei /usr/include/stdio.h, gekürzt

Die Standard-Include-Dateien wie `stdio.h` dürfen in beliebiger Reihenfolge im Programm aufgeführt werden, auch mehrmals. Es dürfen auch zwei Standard-Include-Dateien aufgeführt werden, die beide dasselbe Makro definieren. Eine Standard-Include-Datei schließt niemals eine andere Standard-Include-Datei ein. Wie sich Nichtstandard-Include-Dateien verhalten, ist offen.

1.9.3 Bedingte Kompilation (#ifdef)

Bei der Programmentwicklung möchte man gelegentlich leicht voneinander abweichende Fassungen eines ausführbaren Programms erzeugen, ohne dafür verschiedene Quellfiles schreiben zu müssen. Unser C-Programm 1.89 auf Seite 153 zur Berechnung von Primzahlen hat verschiedene Obergrenzen, je nachdem ob es unter PC-DOS oder UNIX läuft. Im Programmkopf vor `main()` stehen daher folgende Zeilen[34]:

```
#ifdef UNIX
    #define MAX (unsigned long)1000000
#else
    #define MAX (unsigned long)100000
#endif
```

Die symbolische (benamte) Konstante `MAX` soll offenbar auf UNIX-Systemen einen höheren Wert haben als auf PC-DOS-Systemen. Das hängt mit der Speichersegmentierung unter PC-DOS zusammen. Ruft man den Compiler mit einer entsprechenden Option auf:

```
cc -o prim prim.c -DUNIX
```

so wird eine Konstante namens `UNIX` definiert und infolgedessen der if-Zweig vom Präprozessor ausgewertet mit der Folge, daß die Konstante `MAX`, die den untersuchten Zahlenbereich nach oben begrenzt, auf eine Million gesetzt wird. Beim Kompilieren unter PC-DOS entfällt die Option `-DUNIX`.

Eine zweite Anwendung ist die Erzeugung von Programmversionen, die zwecks Fehlersuche etwas gesprächiger sind als die Endfassung. Man gibt beispielsweise Zwischenwerte folgendermaßen aus:

[34]Manche Compiler verlangen, daß das Doppelkreuz in jedem Fall am Beginn der Zeile steht.

```
#define DEBUG 1

int main()
{
...
#if DEBUG
printf("Variable x hat den Wert %d\n", x);
#endif
...
}
```

Hier definieren wir in der Programmquelle ein symbolische Konstante `DEBUG` zu 1 (= true) und veranlassen damit den Präprozessor, die `printf()`-Zeile einzubeziehen. Läuft das Programm fehlerfrei, setzt man im Quellcode `DEBUG` auf null. Ein anderes, im folgenden Programm angewendetes Verfahren fragt nur danach, ob die Konstante `DEBUG` definiert ist oder nicht.

```
/* Programm itox zum Umrechnen von positiven ganzen Zahlen
zur Basis 10 auf eine andere Basis, z. B. 16 */

#define  DEBUG             /* falls gewuenscht */
#define  MAX 8             /* max. Stellen Ergebnis */
#include <stdio.h>

int umwandeln(int, int, int *);         /* Prototyp */

int main()
{
int b;       /* neue Basis, als Dezimalzahl */
int x;       /* umzurechnende Dezimalzahl */
int i;       /* Schleifenzaehler */
int r[MAX];  /* Array des Ergebnisses */

while (1) {
        printf("\n\nNeue Basis:  ");    /* einlesen */
        scanf("%d", &b);
        printf("Dezimalzahl: ");
        scanf("%d", &x);

        if (b < 2) {
                puts("Basis muss > 1 sein.");
                continue;
        }

        if (x < 1) {
                puts("Dezimalzahl muss > 0 sein.");
                continue;
        }

#ifdef DEBUG
printf("\nb = %d  x = %d\n\n", b, x);
#endif

        for (i = 0; i < MAX; i++) {      /* Array nullen */
                r[i] = 0;
        }

        umwandeln(b, x, r);              /* rechnen */

                                         /*ausgeben */
        printf("Die Dezimalzahl %d lautet zur Basis %d: ", x, b);
        for (i = MAX - 1; i >= 0; i-) {
                printf("%d ", r[i]);
        }
```

```
} /* Ende while, Schleife mit control-c verlassen */
return 0;
}

/* Funktion umwandeln() */

int umwandeln(int b, int z, int *r)
{
int i, j, y;

while (z >= 1) {
        y = 1;
        for (i = 0; y <= z; i++) {
                y *= b;
        }
        y = y / b; i-;    /* eins zuviel */

#ifdef DEBUG
printf("y = %d  i = %d\n", y, i);
#endif

        for (j = 0; y <= z; j++) {
                z -= y;
        }

#ifdef DEBUG
printf("z = %d  j = %d\n\n", z, j);
#endif

        r[i] = j;          /* Ausgabe in Array */
}  /* Ende while */
return 0;
}
```

Quelle 1.75 : C-Programm itox.c zur Umrechnung von Dezimalzahlen auf eine andere Basis

1.9.4 Memo Präprozessor

- Im ersten Schritt des Kompiliervorgangs durchläuft die Programmquelle den Präprozessor.
- Der Präprozessor entfernt Kommentar, ersetzt symbolische Konstanten und Makros zeichengetreu, fügt den Inhalt von Include-Dateien ein und berücksichtigt bzw. verwirft Programmzeilen, die bedingt zu kompilieren sind.

1.9.5 Übung Präprozessor

Ergänzen Sie das Programm zur Weganalyse dahingehend, daß es in der DEBUG-Version nach jeder Teilstrecke die Zwischenergebnisse auf dem Bildschirm ausgibt, in der Endversion nur das Gesamtergebnis.

Schreiben Sie ferner alle Stringkonstanten in eine Include-Datei, von der Sie eine deutsche und eine englische oder französische Fassung herstellen. Beim Compiler-Aufruf soll mittels einer Option die Sprache ausgewählt werden.

1.10 Dokumentation

1.10.1 Zweck

Die **Dokumentation** dient dazu, ein Programm im Quellcode einem menschlichen Leser verständlich zu machen. Längere undokumentierte Programme sind nicht nachzuvollziehen.

Eine Dokumentation[35] gehört zu jedem Programm, das länger als eine Seite ist und länger als einen Tag benutzt werden soll.

Andererseits zählt das Schreiben von Dokumentationen nicht zu den Lieblingsbeschäftigungen der Programmierer, das Erfolgserlebnis fehlt. Wir stellen hier einige Regeln auf, die für Programme zum Eigengebrauch gelten; bei kommerziellen Programmen gehen die Forderungen weiter.

Die erste Gelegenheit zum Dokumentieren ist der **Kommentar** im Programm. Man soll reichlich kommentieren, aber keine nichtssagenden Bemerkungen einflechten. Wenn der Kommentar etwa die Hälfte des ganzen Programms ausmacht, ist das noch nicht übertrieben.

1.10.2 Anforderungen (DIN 66 230)

Zur Dokumentation legt die Norm DIN 66 230 *Programmdokumentation* Begriffe und Regeln fest. Eine weitere Norm ist der ANSI/IEEE Std 1063-1987 *Standard for Software User Documentation* in Verbindung mit einer Reihe weiterer IEEE-Standards. Wir verwenden folgende vereinfachte Gliederung:

1. Allgemeines
    - Name des Programms, Programmart (Vollprogramm, Funktion)
    - Zweck des Programms
    - Programmiersprache
    - Computertyp, Betriebssystem
    - Geräte (Drucker, Plotter, Maus)
    - Struktur als Grafik, Fließbild
    - externe Unterprogramme, soweit verwendet
2. Anwendung
    - Aufruf
    - Konstante, Variable
    - Eingabe (von Tastatur, aus Dateien)
    - Ausgabe (zum Bildschirm, Drucker, in Dateien)
    - Einschränkungen
    - Fehlermeldungen
    - Beispiel
    - Speicherbedarf
    - Zeitbedarf
3. Verfahren
    - Algorithmus
    - Genauigkeit
    - Gültigkeitsbereich
    - Literatur zum Verfahren
4. Bearbeiter
    - Name, Datum der Erstellung
    - Name, Datum von Änderungen

[35] Real programmers write programs, not documentation.

Das sieht nach Arbeit aus. Man braucht nicht in allen Fällen alle Punkte zu berücksichtigen, aber ohne eine solche Dokumentation läßt sich ein Programm nicht zuverlässig benutzen und weiterentwickeln. Im Netz finden sich Werkzeuge, die in Verbindung mit dem Kommentar und der Struktur eines Programmes halbautomatisch eine Dokumentation erstellen. Ein Beispiel ist *Doxygen*:

`http://www.doxygen.org/`

Zweckmäßig beschäftigt man sich mit solchen Werkzeugen frühzeitig, nicht erst, wenn der Code fast fertig ist.

1.10.3 Erstellen einer man-Seite

Die inhaltliche Gliederung einer man-Seite wurde bereits im Abschnitt **??** *Wo schlägt man nach?* auf Seite **??** erläutert. Hier geht es um die technische Herstellung. Hilfreich sind die man-Seiten zu `man(1)` und `man(5)`.

Das Kommando `man(1)` sucht die Dokumentationen in den durch die Umgebungs-Variable `MANPATH` bekannten Verzeichnissen. Das ist heute eine lange Aufzählung, oft länger als die unter `PATH`. Vom Benutzer eingerichtete oder erstellte man-Seiten liegen vor allem unter:

- `/usr/local/man/`
- `/usr/share/man/`
- `/opt/*/man/`

In diesen Verzeichnissen finden sich für jede Sektion von 1 bis 9 bis zu vier Arten von Unterverzeichnissen:

- `catx`
- `catx.Z`
- `manx`
- `manx.Z`

wobei für `x` die Sektionsnummer einzusetzen ist. Die Z-Verzeichnisse enthalten die Dokumentationen komprimiert mit `compress(1)` (unter LINUX mit GNU-zip), die beiden anderen Verzeichnisse die unkomprimierten Texte. Die `man`-Verzeichnisse enthalten `nroff(1)`-Quelltexte, die `cat`-Verzeichnisse die formatierten, druckfertigen Texte. Im Verzeichnis `/usr/share/man/man1.Z/` findet sich beispielsweise die Datei `ls.1` mit der komprimierten `nroff`-Quelle der man-Seite zum Kommando `ls(1)`. Gegebenenfalls liegt unter `/usr/share/man/man1/` die entkomprimierte `nroff`-Quelle. Unter `/usr/share/man/cat1.Z` und `/usr/share/man/cat1/` liegen die formatierten Seiten. Mittels:

```
cat /usr/share/man/cat1.Z/ls.1 | uncompress | more
```

könnte man sich eine man-Seite anschauen, mit `man(1)` geht es einfacher.

Will man selbst eine man-Seite schreiben, kopiert man sich am besten eine ähnliche man-Seite:

```
cat /usr/share/man/man1.Z/pwd.1 | uncompress > mycommand.1
```

und editiert diese. Vermutlich muß man sich vorher etwas mit dem `nroff`-Format auseinandersetzen und mit den unter `man(5)` beschriebenen Makros. Die so erzeugte unkomprimierte `nroff(1)`-Quelle gehört in ein `man`-Verzeichnis. Die mittels `compress(1)` verdichtete Qelle kommt ohne die Kennung `.Z` nach `man.Z/`. Dann formatiert man die Quelle:

```
nroff mycommand.1 > mycommand.1.cat
```

und kopiert sie unter dem Namen `mycommand.1` in ein `cat`-Verzeichnis. Schließlich ist die formatierte Datei zu komprimieren und in ein `cat.Z`-Verzeichnis zu stellen. Die Datei heißt in jeder Fassung immer nur `mycommand.1`, das Format ergibt sich aus dem jeweiligen Verzeichnis. Man braucht nicht alle Formate zu erzeugen, eines reicht.

Im WWW hat die *Open Group* unter dem URL:

```
http://www.opengroup.org/common_access/
```

ein umfangreiche Zusammenstellung von man-Seiten samt Suchmaschine veröffentlicht. Eine Suche nach dem Begriff *time* ergab 43 Seiten, von `at(1)` bis `xshrealtime()`. Die Zusammenstellung gehört zur *Single UNIX Specification* und enthält nicht die Seiten von zusätzlichen Programmen wie `sendmail(1)`.

In der GNU-Welt ist für online-Hilfen das Texinfo-Format gebräuchlich, das mittels des Kommandos `info(1)` gelesen wird. Einzelheiten am besten im WWW. Damit kommen wir zu einem dritten Weg für online-Dokumentationen, nämlich HTML-Seiten, die mit einem WWW-Browser gelesen werden. Trotz der erweiterten Möglichkeiten (Unterteilung eines umfangreichen Themas, Hyperlinks, Grafik) dieser neueren Formate sind die dürren man-Seiten immer noch am weitesten verbreitet. Inhaltlich sind sie den Hilfen anderer Betriebssysteme ohnehin haushoch überlegen.

1.11 Weitere C-Programme

1.11.1 Name

Obwohl es aus den bisherigen Beispielen klar geworden sein müßte, weisen wir nochmals darauf hin: Jedes selbständige C-Programm heißt im Quellcode `main()`, ein anderer Programmname kommt – außer im Kommentar – nirgends im Quelltext vor (in FORTRAN oder PASCAL sieht die Sache anders aus). Der Name der Datei, in der der kompilierte Code steht, ist der Name, unter dem das Programm aufgerufen wird.

Der Name der Datei, in der der Quellcode zu finden ist, hat die Kennung `.c`; die meisten Programmierwerkzeuge erwarten das. Die UNIX-Compiler schreiben standardmäßig das kompilierte Programm in eine Datei namens `a.out`; Microsoft Quick-C nimmt den Namen des Quellfiles ohne die Kennung. In beiden Fällen kann man mit der Compiler-Option `-o` für das Ausgabefile einen beliebigen anderen Namen vereinbaren.

1.11.2 Aufbau

Wir kennen nun die Bausteine, aus denen sich ein Programm zusammensetzt. Wie sieht ein vollständiges Programm aus? Zunächst einige Begriffe zum Aufbau von Programmen.

Die kleinste Einheit, die etwas bewirkt, ist die **Anweisung**. Mehrere Anweisungen können zu einem durch geschweifte Klammern zusammengefaßten **Block** vereinigt werden. Nach außen wirkt dieser Block wie eine einzige Anweisung. Der Block ist zugleich die kleinste Einheit für den Geltungsbereich von Variablen.

Mehrere Anweisungen oder Blöcke werden zu einer **Funktion** zusammengefaßt. Die Funktion ist die kleinste kompilierbare Einheit. Eine oder mehrere Funktionen können in einer **Datei** abgelegt sein. Dem Compiler übergibt man im Minimum eine Datei, die eine Funktion enthält. Mehrere Dateien hinwiederum können ein vollständiges, nach dem Kompilieren lauffähiges **Programm** bilden. Erinnern Sie sich an das Werkzeug `make(1)`?

Das Minimalprogramm in C und C++ besteht aus einer Funktion – nämlich `main()` – deren Rumpf leer ist, in einer Datei:

```
main()
{
}
```

Quelle 1.76 : Minimales C-Programm

Der Name `main()` ist für das **Hauptprogramm** vorgeschrieben. `main()` ist eine Funktion, daher die beiden runden Klammern. Der durch die geschweiften Klammern umschlossene Block ist leer, `main()` tut nichts. Der Syntaxprüfer `lint(1)` beanstandet, daß der Rückgabewert von `main()` undefiniert ist, was stimmt, uns aber nicht weiter stört. Der Compiler erzeugt aus diesem Quellcode etwa 16 kB ausführbaren Code. Hinter `main()` steckt einiges, was dem Programmierer verborgen ist.

Als nächstes wollen wir `main()` als ganzzahlig deklarieren, für einen definierten **Rückgabewert** sorgen und – wie es sich gehört – mittels eines **Kommentars** das Verständnis erleichtern:

```
/* Dies ist ein einfaches C-Programm von hohem
   paedagogischen, aber sonst keinem Wert. */

int main()
{
return 255;    /* 255 groesster zulaessiger Wert */
}
```

Quelle 1.77 : C-Programm, einfachst

Dieses Programm wird vom `lint(1)` gutgeheißen. Die Deklaration von `main()` als `int` könnte entfallen, da sie unabänderlich ist, aber wir wollen uns angewöhnen, alle Größen ausdrücklich zu deklarieren. Den Rückgabewert können Sie sich mit dem Shell-Kommando `print $?` nach der Ausführung anschauen.

Um etwas Vernünftiges zu tun, muß das Programm um einige Zeilen angereichert werden. Wir deklarieren eine `int`-Variable namens `i`, weisen ihr einen Wert zu (definierende Deklaration) und verwenden die C-Standardfunktion `printf(3)` für die Ausgabe auf `stdout`, den Bildschirm. `printf(3)` erwartet als erstes und notwendiges Argument einen Formatstring, dessen reichhaltige Syntax Sie im Referenz-Handbuch finden.

```
/* Dies ist ein einfaches C-Programm von hohem
   paedagogischen, aber sonst fast keinem Wert. */

int main()
{
int i = 53;
printf("\nDer Wert betraegt %d\n", i);
return i;
}
```

Quelle 1.78 : C-Programm, einfach

Nun soll auch der Präprozessor Arbeit bekommen. Wir definieren eine symbolische Konstante `NUMBER` und schließen vorsichtshalber die Include-Datei `stdio.h` ein, die man immer braucht, wenn es um Ein- oder Ausgabe geht. Weiterhin verwenden wir einen arithmetischen Operator und eine Zuweisung:

```
/* Dies ist ein fortgeschrittenes C-Programm  */

#define NUMBER 2

#include <stdio.h>

int main()
{
int i = 53, x;

x = i + NUMBER;
printf("\nDer Wert betraegt %d\n", x);
return 0;
}
```

Quelle 1.79 : C-Programm, fortgeschritten

Da ein Ausdruck sein Ergebnis zurückgibt, können wir in der Funktion `printf(3)` anstelle von x auch die Summe hinschreiben. Als Rückgabewert unseres Hauptprogrammes wollen wir den Rückgabewert der Funktion `printf(3)` haben, nämlich die Anzahl der ausgegebenen Zeichen. Das Programm wird kürzer, aber auch schwieriger zu verstehen (falls man nicht ein alter C-Hase ist):

```
/* Dies ist ein kleines C-Programm  */

#define NUMBER 2

#include <stdio.h>

int main()
{
int i = 53;

return (printf("\nDer Wert betraegt %d\n", i + NUMBER));
}
```

Quelle 1.80 : C-Programm, fortgeschritten, Variante

Der ausführbare Code ist damit auf 35 KB angewachsen. Jetzt wollen wir die beiden Summanden im Dialog erfragen und die Summe als Makro schreiben. Außerdem soll die Rechnung wiederholt werden, bis wir eine Null eingeben:

```
/* Endlich mal was Vernuenftiges */

#define SUMME(x, y) (x + y)

#include <stdio.h>

int main()
{
int a = 1, b, i = 0;

while (a != 0) {
        printf("Ersten Summanden eingeben : ");
        scanf("%d", &a);
        printf("Zweiten Summanden eingeben: ");
        scanf("%d", &b);
        printf("Die Summe ist %d\n", SUMME(a, b));
        i++;
}

return i;
}
```

Quelle 1.81 : C-Programm mit Eingabe

Der Rückgabewert ist die Anzahl der Schleifendurchläufe. Die Stringkonstanten werden nicht mit `puts(3)` ausgegeben, da diese Funktion einen hier unerwünschten Zeilenvorschub anfügt. Denken Sie daran, daß die Funktion `scanf(3)` Pointer als Argumente braucht!

1.11.3 Fehlersuche

Einige Gesichtspunkte sind bereits im Abschnitt 1.2.4 *Debugger* auf Seite 23 behandelt worden. Der erfahrene Programmierer unterscheidet sich vom Anfänger in drei Punkten:

- Er macht raffiniertere Fehler.
- Er weiß das.

- Er kennt die Wege und Werkzeuge zur Fehlersuche.

Fehlerfreie Programme schreibt auch der beste Programmierer nicht. Deshalb ist es wichtig, schon beim Programmentwurf an die Fehlersuche zu denken und vor allem das Programm so zu gestalten, daß es bei einem Fehler nicht ein richtiges Ergebnis vortäuscht. Das ist so ungefähr das Schlimmste, was ein Programm machen kann. Dann lieber noch ein knallharter Absturz. Besser ist eine sanfte Notlandung mit einer aussagekräftigen Fehlermeldung.

Die Programmeinheiten (Funktionen) lasse man nicht zu umfangreich werden. Ein bis zwei Seiten Quelltext überschaut man noch, wird es mehr, sollte man die Funktion unterteilen. Weiterhin gebe man im Entwicklungsstadium an kritischen Stellen die Werte mittels `printf()` oder `fprintf()` aus. Diese Zeilen kommentiert man später aus oder klammert sie gleich in `#ifdef`- und `#endif`-Anweisungen ein. Bewährt hat sich auch, die eigenen Programme einem anderen zu erklären, da wundert man sich manchmal über den eigenen Code. Ein Programm, das man nach ein bis zwei Wochen Pause selbst nicht mehr versteht, war von vornherein nicht gelungen.

Und wenn dann der Computerfreak zu nächtlicher Stunde den Bugs hinterherjagt, schließt sich ein weiter Bogen zurück in die Kreidezeit, denn die ersten Säugetiere – Zeitgenossen der Saurier – waren auch nachtjagende Insektenfresser.

1.11.4 Optimierung

Das erste und wichtigste Ziel beim Programmieren ist – nächst der selbstverständlichen, aber unerreichbaren **Fehlerfreiheit** – die **Übersichtlichkeit**. Erst wenn ein Programm sauber läuft, denkt man über eine **Optimierung** nach. Optimieren heißt schneller machen und Speicher einsparen, sowohl beim Code wie auch zur Laufzeit. Diese beiden Ziele widersprechen sich manchmal. Im folgenden findet man einige Hinweise, die teils allgemein, teils nur für C gelten.

Die optimierende **Compiler-Option** `-O` ist mit Vorsicht zu gebrauchen. Es kommt vor, daß ein optimiertes Programm nicht mehr läuft. Die Gewinne durch die Optimierung sind auch nur mäßig. Immerhin, der Versuch ist nicht strafbar.

Als erstes schaut man sich die **Schleifen** an, von geschachtelten die innersten. Dort sollte nur das Allernotwendigste stehen.

Bedingungen sollten so einfach wie möglich formuliert sein. Mehrfache Bedingungen sollten darauf untersucht werden, ob sie durch einfache ersetzt werden können. Schleifen sollten möglichst dadurch beendet werden, daß die Kontrollvariable den Wert Null und nicht irgendeinen anderen Wert erreicht. Kontrollvariable können auch heruntergezählt werden statt herauf.

Eine Bedingung mit mehreren *ands* oder *ors* wird so lange ausgewertet, bis die Richtigkeit oder Falschheit des gesamten Ausdrucks erkannt ist. Sind mehrere Bedingungen durch *or* verbunden, wird die Auswertung nach Erkennen des ersten richtigen Gliedes abgebrochen. Zweckmäßig stellt man das Glied, das am häufigsten richtig ist, an den Anfang. Umgekehrt ist ein Ausdruck mit durch *and* verbundenen Gliedern falsch, sobald ein Glied falsch ist. Das am häufigsten falsche Glied gehört an den Anfang.

Ähnliche Überlegungen gelten für die `switch`-Anweisung. Die häufigste Auswahl sollte als erste abgefragt werden. Ist das der `default`-Fall, kann er durch eine eigene `if`-Abfrage vor der Auswahl abgefangen werden.

Überflüssige **Typumwandlungen** – insbesondere die unauffälligen impliziten – sollten zumindest in Schleifen vermieden werden. Der Typ numerischer Konstanten sollte von vornherein zu den weiteren Operanden passen. Beispielsweise führt

```
float f, g;
g = f + 1.2345;
```

zu einer Typumwandlung von f in double und einer Rückwandlung des Ergebnisses in float, da Gleitkommakonstanten standardmäßig vom Typ double sind.

Gleitkommarechnungen sind aufwendiger als Rechnungen mit ganzen Zahlen und haben zudem noch Tücken infolge von Rundungsfehlern. Eine Gleitkomma-Null ist nicht immer

wirklich null. Wer mit Geldbeträgen rechnet, sollte mit ganzzahligen Centbeträgen anstelle von gebrochenen Eurobeträgen arbeiten. Wenn schon mit Gleitkommazahlen gerechnet werden muß und der Speicher ausreicht, ist der Typ `double` vorzuziehen, der intern ausschließlich verwendet wird.

Von zwei möglichen **Operationen** ist immer die einfachere zu wählen. Beispielsweise ist eine Addition einfacher als eine Multiplikation, eine Multiplikation einfacher als eine Potenz und eine Bitverschiebung einfacher als eine Multiplikation mit 2. Eine Abfrage

```
if (x < sqrt(y))
```

schreibt man besser

```
if (x * x < y)
```

Manchmal kann man durch Umstellen einer Formel die Anzahl der Rechenschritte verringern. Ob man mehrfach benötigte Zwischenergebnisse speichert oder besser jedesmal neu berechnet, ist nicht allgemein zu entscheiden.

Kleine Funktionen lassen sich durch **Makros** ersetzen, die vom Präprozessor in In-line-Code umgewandelt werden. Damit erspart man sich den Funktionsaufruf samt Parameterübergabe. Der ausführbare Code wird geringfügig länger.

Enthält eine Schleife nur einen Funktionsaufruf, ist es besser, die Schleife in die Funktion zu verlegen, da jeder Funktionsaufruf Zeit kostet.

Die maßvolle Verwendung **globaler Variabler** verbessert zwar nicht den Stil, aber die Geschwindigkeit von Programmen mit vielen Funktionsaufrufen, da die Parameterübergabe entfällt.

Die Verwendung von **Bibliotheksfunktionen** kann in oft durchlaufenen Schleifen stärker verzögern als der Einsatz spezialisierter selbstgeschriebener Funktionen, da Bibliotheksfunktionen allgemein und kindersicher sind. Verwenden Sie die einfachste Funktion, die den Zweck erfüllt, also `puts(3)` anstelle von `printf(3)`, wenn es nur um die Ausgabe eines Strings samt Zeilenwechsel geht.

Die Adressierung von Arrayelementen durch Indizes ist langsamer als die Adressierung durch **Pointer**. Der Prozessor kennt nur Adressen, also muß er Indizes erst in Adressen umrechnen. Das erspart man ihm, wenn man gleich mit Adressen sprich Pointern arbeitet. Wer die Pointerei noch nicht gewohnt ist, schreibt das Programm zunächst mit Indizes, testet es aus und stellt es dann auf Pointer um. Ein Beispiel:

```
long i, j, a[32];
/* Adressierung durch Indizes, langsam */
a[0] = a[i] + a[j];
/* Adressierung durch Pointer, schnell */
*a = *(a + i) + *(a + j);
```

Wir erinnern uns, der Name eines Arrays ist der Pointer auf das erste Element (mit dem Index 0). Experimente mit verschiedenen Compilern haben allerdings den Verdacht geschürt, dass manche Compiler von sich aus die schnellere Variante (Pointer) wählen. Übergeben Sie große Strukturen als Pointer, nicht als Variable. Dadurch spart man das Kopieren der Daten.

Input/Output ist immer zeitaufwendig. In vielen Programmen geht für diese Aufgabe sogar der größte Teil der Zeit drauf. Also unnötigen I/O vermeiden.

Den größten Gewinn an Geschwindigkeit und manchmal auch zugleich an Speicherplatz erzielt man durch eine zweckmäßige **Datenstruktur** und einen guten **Algorithmus**. Diese Überlegungen gehören jedoch an den Anfang der Programmentwicklung.

Wenn das alles noch nicht reicht, ist es Zeit, sich mittels eines Profilers anzusehen, wo die Zeit verbraten wird, und vielleicht auch einige kritische Programmteile in Assembler schreiben.

1.11.5 curses – Fluch oder Segen?

Im Englischen ist ein *curse* so viel wie ein Fluch, und die `curses(3)`-Bibliothek ist früher wegen ihrer vielen Fehler oft verwünscht worden. Andererseits erleichtert sie den Umgang mit dem Terminal unabhängig von dessen Typ. Wir beginnen mit einem einfachen

Programm, das `terminfo`-Funktionen aus der `curses(3)`-Bibliothek verwendet, um den Bildschirm zu löschen, wobei der Terminaltyp aus der Umgebungsvariablen `TERM` und die zugehörigen Steuersequenzen aus der Terminalbeschreibung in `/usr/lib/terminfo(4)` entnommen werden. Das Programm soll außerdem, wenn ihm Dateinamen als Argumente übergeben werden, die Dateien leeren, ohne sie zu löschen (der Bildschirm wird ja auch nicht verschrottet):

```
/* C-Programm, das Bildschirm oder Files loescht */
/* Compile: cc -o xclear xclear.c -lcurses */
/* falls terminfo-Fkt. verwendet werden sollen, noch
   -DTERMINFO anhaengen */

#include <curses.h>        /* enthaelt stdio.h */

#ifdef TERMINFO
#include <term.h>          /* nur fuer terminfo */
#endif

int main(argc, argv)

int argc;
char *argv[];

{
int i;

if (argc > 1) {      /* Files leeren, nicht loeschen */

    for(i = 1; i < argc; i++) {
        if (!access(argv[i], 0))
            close(creat(argv[i], 0));
        else
            printf("File %s unzugaenglich.\n", argv[i]);
    }
}

else {

#ifdef TERMINFO    /* Bildschirm leeren, terminfo */

    setupterm(0, 1, 0);
    putp(clear_screen);
    resetterm();

#else              /* Bildschirm leeren, curses */

    initscr();
    refresh();
    endwin();

#endif

}
return 0;
}
```

Quelle 1.82 : C-Programm zum Leeren des Bildschirms oder von Dateien

Das Kommando `/usr/local/bin/xclear` ist eine recht praktische Erweiterung von `/bin/clear`. Die Funktion `setupterm()` ermittelt vor allem den Terminaltyp. `putp()` schickt die Steuersequenz zum Terminal, und `reset_shell_mode()` bereinigt alle Dinge, die `setupterm()` aufgesetzt hat. Mit diesen `terminfo`-Funktionen soll man nur in einfachen Fällen wie dem obigen arbeiten, in der Regel sind die intelligenteren `curses(3)`-Funktionen vorzuziehen.

Im folgenden Beispiel verwenden wir `curses(3)`-Funktionen zur Bildschirmsteuerung zum Erzeugen eines Hilfe-Fensters:

```
/* help.c, Programm mit curses-Funktionen */
/* Compiler: cc -o help help.c -lcurses */

#define END  ((c == 'Q') | (c == 'q'))   /* Makros */
#define HELP ((c == 'H') | (c == 'h'))

#include <curses.h>

int main()
{
int    c, disp=1;
WINDOW *frame;

 initscr();
 noecho();
 cbreak();
 mvprintw(10,15,"Program demonstrating Curses-Windows");
 mvprintw(11,15,"You get a help-window by pressing h");
 mvprintw(LINES-1,0,"Press q to quit");
 refresh();
 while (1) {
  c = getch();
  if END {
   clear();
   refresh();
   endwin();
   return 0;
  }
  else
   if HELP {
    if (disp) {
     frame = newwin(13,40,10,35);
     wstandout(frame);
     for (c = 0; c <= 4; c++)
      mvwprintw(frame,c,0,"%42c",' ');
     mvwprintw(frame,5,0,"Window built by curses. It may");
     mvwprintw(frame,6,0,"contain helpful messages.     ");
     mvwprintw(frame,7,0,"Delete the window by typing h.");
     for (c = 8; c <= 12; c++)
      mvwprintw(frame,c,0,"%42c",' ');
     wrefresh(frame);
     wstandend(frame);
    }
    else {
     delwin(frame);
     touchwin(stdscr);
     refresh();
    }
   disp = !disp;
  }
 }
}
```

Quelle 1.83 : C-Programm mit curses-Funktionen

Jedes `curses`-Programm muss die Include-Datei `curses.h` enthalten, die seinerseits `stdio.h` einschließt. `WINDOW` ist ein in `curses.h` definierter Datentyp, eine Struktur, die den Bildschirminhalt, die Cursorposition usw. enthält. Die `curses(3)`-Funktionen bewirken folgendes:

- `initscr()` muß die erste `curses(3)`-Funktion sein. Sie initialisiert die Datenstruktu-

ren. Das Gegenstück dazu ist `endwin()`, die das Terminal wieder in seinen ursprünglichen Zustand versetzt.

- `noecho()` schaltet das Echo der Tastatureingaben auf dem Bildschirm aus.
- `cbreak()` bewirkt, daß jedes eingegebene Zeichen sofort an das Programm weitergeleitet wird – ohne RETURN.
- `mvprintw()` bewegt (move) den Cursor an die durch die ersten beiden Argumente bezeichnete Position (0, 0 links oben) und schreibt dann in das Standardfenster (sofern nicht anders angegeben). Syntax wie die C-Standardfunktion `printf(3)`.
- `refresh()` Die bisher aufgerufenen Funktionen haben nur in einen Puffer geschrieben, auf dem tatsächlichen Bildschirm hat sich noch nichts gerührt. Erst mit `refresh()` wird der Puffer zum Bildschirm übertragen.
- `getch()` liest ein Zeichen von der Tastatur
- `clear()` putzt den Bildschirm beim nächsten Aufruf von `refresh()`
- `newwin()` erzeugt ein neues Fenster – hier mit dem Namen `frame` – auf der angegebenen Position mit einer bestimmten Anzahl von Zeilen und Spalten.
- `wstandout()` setzt das Attribut des Fensters auf `standout`, d. h. auf umgekehrte Helligkeiten beispielsweise. Gilt bis `wstandend()`.
- `wrefresh()` wie `refresh()`, nur für ein bestimmtes Fenster.
- `delwin()` löscht ein Fenster, Gegenstück zu `newwin()`.
- `touchwin()` schreibt beim nächsten `refresh()` ein Fenster völlig neu.

Die `curses(3)`-Funktionen machen von den Terminalbeschreibungen in den `/usr/lib/terminfo`-Dateien Gebrauch, man braucht sich beim Programmieren um den Terminaltyp nicht zu sorgen. Andererseits kann man nichts verwirklichen, was in `terminfo` nicht vorgesehen ist, Grafik zum Beispiel.

1.11.6 Mehr oder weniger zufällig

Man braucht im Leben manchmal **Zufallszahlen** (random number), zum Beispiel wenn eine Reihe von Vokabeln bunt durcheinander abgefragt werden soll. Sind die Anforderungen nicht so hoch geschraubt, daß man erst einmal ein Philosophisches Seminar über den Begriff *Zufall* absolvieren muß, reichen einige Funktionen aus der C-Standard-Bibliothek. Das folgende Programm erzeugt eine Folge mehr oder weniger zufälliger natürlicher Zahlen und gibt sie auf `stdout` aus:

```
/* random.c zur Erzeugung von Zufallszahlen
   MAX Zufallszahlen von 1 bis MOD */

#define MAX 100
#define MOD 200

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main()
{

int i, r, s;

s = (int) time((time_t *) 0);

/*
srand((unsigned) s);
for (i = 0; i < MAX; i++) {
```

```
r = 1 + (rand() % MOD);
printf("%d\n", r);
}
*/

srandom((unsigned) s);
for (i = 0; i < MAX; i++) {
r = 1 + (random() % MOD);
printf("%d\n", r);
}

return 0;
}
```

Quelle 1.84 : C-Programm zur Erzeugung von MAX Zufallszahlen im Bereich von 1 bis MOD

Wir definieren die Anzahl MAX von Zufallszahlen, die wir brauchen, sowie den Modulus MOD, der die größte Zufallszahl bestimmt. Weiter binden wir die Include-Dateien stdio.h für die Funktion printf(), time.h für den Systemaufruf time() und stdlib.h für die Funktionen srandom() und random() ein. Die Variable i ist ein Schleifenzähler, r die jeweilige Zufallszahl und s der sogenannte Seed (Samen), auch Salz genannt, den wir brauchen, um den Zufallszahlengenerator zu starten.

Den **Seed** gewinnen wir aus der Systemuhr als Anzahl der Sekunden seit dem 1. Januar 1970, 0 Uhr GMT. Damit ist sichergestellt, daß wir bei jedem Aufruf einen anderen Wert haben. Die Syntax von time() holt man sich mittels man 2 time. Das Argument von time() ist hier der Nullpointer.

Die Funktion srand() oder srandom() startet den Generator rand() beziehungsweise random. Beide Funktionspaare verwenden unterschiedliche Algorithmen, siehe die zugehörigen man-Seiten. Das Ergebnis des Generators wird modulo MOD genommen, um den Zahlenbereich zu begrenzen. Da wir Zahlen von 1 bis MOD, die Grenzen eingeschlossen, haben wollen, addieren wir eine 1 hinzu. Das Ergebnis dieser Rechnung wird wie üblich mittels der Funktion printf() auf stdout ausgegeben.

Wir wollen nun das Programm so umstricken, daß die Rechnung in einer Funktion zufallszahl() durchgeführt wird und das Hauptprogramm nur die Zahlen ausgibt:

```
/* random2.c zur Erzeugung von Zufallszahlen
   MAX Zufallszahlen von 1 bis MOD */

#define MAX 100
#define MOD 200

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int zufallszahl(int m);       /* Prototyp */

int main()
{
int i;

for (i = 0; i < MAX; i++) {
printf("%d\n", zufallszahl((int) MOD));
}

return 0;
}

/* Funktion zufallszahl() */

int zufallszahl(int m)
{
```

```
static int r = 0;
unsigned s;

if (!r) {
        s = (unsigned) time((time_t *) 0);
        srand(s);
}

r = 1 + (rand() % m);

return r;
}
```

Quelle 1.85 : C-Programm zur Erzeugung von MAX Zufallszahlen im Bereich von 1 bis MOD, mit Funktion

Um die Funktion `zufallszahl()` allgemein verwendbar zu gestalten, übergeben wir den Modulus bei jedem Aufruf als Argument. Da der Generator nur beim ersten Aufruf gestartet werden soll, deklarieren wir die Variable `r` als `static`, initialisieren sie mit null und mißbrauchen sie als Flag für den Generatorstart mittels `srand()`. Die Initialisierung wird nur einmal, beim Programmaufruf, ausgeführt. Danach hat `r` immer den jeweils jüngsten Zufallswert, der minimal 1 ist und negiert stets `false` liefert.

Im wirklichen Leben verlangte die Aufgabe eine C-Funktion für ein PASCAL-Programm. Hierzu müssen Funktion und Hauptprogramm in getrennten Dateien vorliegen, da es keine zweisprachigen Compiler gibt. Also wurden die C-Funktion isoliert und ein Rahmenprogramm zum Testen in PASCAL geschrieben:

```
/* Funktion zufallszahl.c zur Erzeugung von Zufallszahlen
   MAX Zufallszahlen von 1 bis MOD */

#include <time.h>
#include <stdlib.h>

int zufallszahl(int m)
{

static int r = 0;
unsigned s;

if (!r) {
        s = (unsigned) time((time_t *) 0);
        srand(s);
}

r = 1 + (rand() % m);

return r;
}
```

Quelle 1.86 : C-Funktion zur Erzeugung einer Zufallszahl im Bereich von 1 bis MOD

Die C-Funktion gibt pro Aufruf eine Zufallszahl im Bereich von 1 bis zum Modulus zurück, der als Argument übergeben wird.

```
{PASCAL-Programm, das C-Funktion aufruft}
{Compileraufruf pc -o prandom prandom.p zufallszahl.o}
{Funktion zufallszahl() braucht als Argument den Modulus}

program prandom (input, output);
```

```
var a, i, x: integer;

function zufallszahl(x: integer): integer;
         external C;

begin
writeln('Bitte Modulus eingeben!');
readln(x);
writeln('Bitte Anzahl eingeben!');
readln(a);

writeln('Zufallszahlen:');
for i := a downto 1 do
        writeln(zufallszahl(x));
end.
```

Quelle 1.87 : PASCAL-Programm zur Erzeugung von Zufallszahlen im Bereich von 1 bis MOD, mit C-Funktion

Die Compileraufrufe lauten:

```
cc -c zufallszahl.c
pc -o prandom prandom.p zufallszahl.o
```

Die gemischte Programmierung funktioniert hier reibungslos, weil beide Sprachen den Typ Ganzzahl kennen und die C-Funktion einfach ist.

Wie wir anfangs auf Seite **??** bemerkt haben, ist es prinzipiell unmöglich, mit einer deterministischen Maschine Zufallsergebnisse zu erzeugen. Die vorstehenden Programme liefern daher auch nur Pseudo-Zufallszahlen (pseudo random number), die hinsichtlich bestimmter Eigenschaften mit echten Zufallszahlen übereinstimmen. Für viele Zwecke reicht das, für einen Vokabeltrainer sicherlich. Eine ausführliche Diskussion findet sich bei DONALD E. KNUTH.

1.11.7 Ein Herz für Pointer

Pointer sind nicht schwierig, sondern allenfalls gewöhnungsbedürftig. Sie sind bei C-Programmierern beliebt, weil sie zu eleganten und schnellen Programmen führen. Wir wollen uns an Hand einiger Beispiele an ihren Gebrauch gewöhnen. Eine Wiederholung:

- Der Computer kennt nur Speicherplätze in Einheiten von einem Byte. Jedes Byte hat eine absolute Adresse (Hausnummer), die uns aber nichts angeht.
- Die Deklaration einer Variablen erzeugt eine Variable mit einem Namen und bestimmten Eigenschaften, darunter den durch den Typ bestimmten Speicherbedarf in Bytes.
- Die Definition einer Variablen weist ihr einen Wert zu, belegt Speicherplatz und damit eine Adresse.
- Der Pointer auf eine Variable enthält ihre Speicheradresse. Da uns der absolute Wert der Adresse nicht interessiert, greifen wir auf den Pointer mittels seines Namens zu. Heißt die Variable `x`, so ist `&x` der Name des Pointers.
- Deklariert man zuerst den Pointer `px`, so erhält man die Variable durch Dereferenzierung `*px`. Es ist nicht immer gleichgültig, ob man den Pointer oder die Variable deklariert und das Gegenstück durch Referenzieren bzw. Dereferenzieren handhabt.
- Eine Variable kann notfalls auf einen Namen verzichten, aber niemals auf ihren Pointer.
- Pointer sind keine ganzen Zahlen (die Arithmetik läuft anders).
- Ein Pointer auf eine noch nicht oder nicht mehr existierende Variable hängt in der Luft (dangling pointer) und ist ein Programmfehler.

Nun einige Beispiele zu bestimmten Anwendungen von Pointern.

1.11.7.1 Nullpointer

Die Zahl Null ist die einzige Konstante, die sinnvollerweise einem Pointer zugewiesen werden kann. Auf dieser Adresse liegt kein gültiges Datenobjekt, sie tritt nur in Verbindung mit Fehlern oder Ausnahmen auf. Um die Besonderheit dieser Adresse hervorzuheben, schreibt man sie meist nicht als Ziffer, sondern als Wort *NULL*. In der Include-Datei `stdio.h` ist das Wort als symbolische Konstante mit dem Wert 0 definiert.

Im Programm 1.94 *Sortieren nach Duden* auf Seite 162 kommen beim Öffnen einer Datei zum Lesen folgende Zeilen vor:

```
if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("Datei %s kann nicht goeffnet werden.\n", argv[1]);
    exit(1);
}
```

Die Funktion `fopen()`, die die Datei öffnet, gibt bei Mißerfolg – aus welchen Gründen auch immer – anstatt eines Dateipointers den Nullpointer zurück. Falls der Vergleich positiv ausfällt, also bei Mißerfolg, wird eine Fehlermeldung ausgegeben und das Programm mit dem Systemaufruf `exit()` verlassen. Bei Erfolg enthält der Pointer `fp` die Adresse des Dateianfangs.

1.11.7.2 Pointer auf Typ void

Man braucht gelegentlich einen Pointer, der auf eine Variable von einem zunächst noch unbekannten Typ zeigt. Wenn es dann zur Sache geht, legt man den Typ mittels des `cast`-Operators fest.

Früher nahm man dafür Pointer auf den Typ `char`, denn dieser Typ belegt genau ein Byte, woraus man jeden anderen Typ aufbauen kann. Nach ANSI ist hierfür der Typ `void` zu wählen. Jeder Pointer kann ohne Verlust an Information per `cast` in einen Pointer auf `void` verwandelt werden, und umgekehrt. Die Pointer belegen ja selbst – unabhängig vom Typ, auf den sie zeigen – gleich viele Bytes.

Im folgenden Beispiel wird eine Funktion `xread()` vorgestellt, die jede Tastatureingabe als langen String übernimmt und dann die Eingabe – erforderlichenfalls nach Prüfung – in einen gewünschten Typ umwandelt. Die Funktion ist ein Ersatz für `scanf(3)` mit der Möglichkeit, fehlerhafte Eingaben nach Belieben zu behandeln. Als erstes ein Programmrahmen, der die Funktion `xread()` aufruft, dann die Funktion:

```
/* Fkt. xread() zum Einlesen und Umwandeln von Strings */
/* mit Rahmenprogramm main() zum Testen, 1992-05-11 */

#include <stdio.h>

int xread(void *p, char *typ);
void exit();            /* Systemaufruf */

int main()
{
int error = 0;
int x;
double y;
char z[80];

/* Integer-Eingabe */

printf("Bitte Ganzzahl eingeben: \n");
if (!xread(&x, "int")) {
    printf("Die Eingabe war: %d\n", x);
}
else {
    puts("Fehler von xread()");
    error = 1;
}
```

```
/* Gleitkomma-Eingabe */

printf("Bitte Gleitkomma-Zahl eingeben: \n");
if (!xread(&y, "float")) {
    printf("Die Eingabe war: %f\n", y);
}
else {
    puts("Fehler von xread()");
    error = 1;
}

/* Stringeingabe */

printf("Bitte String eingeben: \n");
if (!xread(z, "char")) {
    printf("Die Eingabe war: %s\n", z);
}
else {
    puts("Fehler von xread()");
    error = 1;
}
exit(error);
}

/* Funktion xread() */
/* Parameter: Variable als Pointer, C-Typ als String */

#define MAXLAENGE 200   /* max. Laenge der Eingabe */
#include <string.h>

int atoi();          /* Standard-C-Bibliothek */
long atol();         /* Standard-C-Bibliothek */
double atof();       /* Standard-C-Bibliothek */

int xread(p, typ)
    void *p;
    char *typ;
{
char input[MAXLAENGE];
int rwert = 0;

if (gets(input) != NULL) {
    switch(*typ) {
        case 'c':       /* Typ char */
            strcpy((char *)p, input);
            break;
        case 'i':       /* Typ int */
        case 's':       /* Typ short */
            *((int *)p) = atoi(input);
            break;
        case 'l':       /* Typ long */
            *((long *)p) = atol(input);
            break;
        case 'd':       /* Typ double */
        case 'f':       /* Typ float */
            *((double *)p) = atof(input);
            break;
        default:
            puts("xread: Unbekannter Typ");
            rwert = 1;
    }
}
else {
    puts("xread: Fehler bei Eingabe");
    rwert = 2;
```

```
}
return rwert;
}
```

Quelle 1.88 : C-Programm mit Pointer auf void

Die Funktion `xread()` braucht als erstes Argument einen Pointer (aus demselben Grund wie `scanf(3)`, call by reference) auf die einzulesende Variable, als zweites Argument den gewünschten Typ in Form eines Strings. Auf eine wechselnde Anzahl von Argumenten verzichten wir hier.

Falls wir nicht für jeden einzulesenden Typ eine eigene Funktion schreiben wollen, muß `xread()` einen Pointer auf einen beliebigen Typ, sprich `void`, übernehmen. Erst nach Auswertung des zweiten Argumentes weiß `xread()`, auf was für einen Typ der Pointer zeigt.

Das Programm läuft in seiner obigen Fassung einwandfrei, der Syntax-Prüfer `lint(1)` hat aber einige Punkte anzumerken.

1.11.7.3 Arrays und Pointer

Das folgende Programm berechnet die **Primzahlen** von 2 angefangen bis zu einer oberen Grenze, die beim Aufruf eingegeben werden kann. Ihr Maximalwert hängt verständlicherweise vom System ab. Aus Geschwindigkeitsgründen werden reichlich Pointer verwendet. Ursprünglich wurden die Elemente der Arrays über Indizes angesprochen, was den Gewohnheiten entgegenkommt. Bei der Optimierung wurden alle Indizes durch Pointer ersetzt, wie im Abschnitt 1.11.4 *Optimierung* auf Seite 140 erläutert.

```
/* Programm zur Berechnung von Primzahlen, 1990-10-03 */
/* Compileraufruf MS-DOS/QuickC: qcl prim.c */
/* Compileraufruf unter UNIX: cc -o prim prim.c -DUNIX */

/* Die groesste zu untersuchende Zahl wird unter MS-DOS
   durch die Speichersegmentierung bestimmt. Kein Daten-
   segment p[] darf groesser als 64 KB sein. Damit liegt
   MAX etwas ueber 150000.
   Unter UNIX begrenzt der verfuegbare Speicher die
   Groesse. Der Datentyp unsigned long geht in beiden
   Faellen ueber 4 Milliarden. */

#ifdef UNIX
    #define MAX (unsigned long)1000000
#else
    #define MAX (unsigned long)100000
#endif

#define MIN (unsigned long)50
                       /* Defaultwert fuer Obergrenze */
#define DEF (unsigned long)10000

#include <stdio.h>

/* globale Variable */

unsigned long p[MAX/10]; /* Array der Primzahlen */
unsigned d[MAX/1000];    /* Haeufigkeit der Differenzen */
unsigned long h[2][11];  /* Haeufigkeit der Primzahlen */
unsigned long z = 1;     /* aktuelle Zahl */
unsigned n = 1;          /* lfd. Anzahl Primzahlen - 1 */

/* Funktionsprototypen */

void ttest();            /* Funktion Teilbarkeitstest */
long time();             /* Systemaufruf zur Zeitmessung */

int main(int argc,char *argv[])       /* Hauptprogramm */
```

```
{
int r;
int i = 1, j, k;
unsigned long ende = DEF;
unsigned long *q;
unsigned long dp, dmax = 1, d1, d2;
unsigned long g;
long zeit1, zeit2, zeit3;

/* Auswertung der Kommandozeile */
/* dem Aufruf kann als Argument die Obergrenze
   mitgegeben werden */
/* keine Pruefung auf negative Zahlen oder Strings */

if (argc > 1) {
    sscanf(*(argv + 1), "%lu", &ende);

        if (ende > MAX) {
        printf("\nZ. zu gross! Maximal %lu\n", MAX);
        exit(1);
    }
        if (ende < MIN) {
        printf("\nZ. zu klein; genommen wird %lu\n\n\n", \
                                                  MIN);
        ende = MIN;
    }
    if (g = ende % 10) {
    printf("\nZ. muss durch 10 teilbar sein: %lu\n\n\n", \
                                        ende=ende - g);
    }

}

/* Algorithmus */

time(&zeit1);

*p = 2; *(p + 1) = 3;          /* die ersten Primzahlen */
ende -= 3;

while (z < ende) {
    z += 4;
    ttest();
    z += 2;
    ttest();
}

/*
   Da z pro Durchlauf um 6 erhoeht wird, kann eine
   Primzahl zuviel berechnet werden,
   gegebenenfalls loeschen
*/

if (*(p + n) > (ende = ende + 3))
    n -= 1;

/* Berechnung der Haeufigkeit in den Klassen */

g = ende/10; **h = 1; **(h + 1) = 0; j = 1; k = 0;

for (i = 0; i <= n; i++) {
    if (*(p + i) > g) {
        *(*h + j) = g;
        *(*(h + 1) + j) = i - k;
        k = i;
```

```
        j++;
        g += ende/10;
    }
}

*(*h + j) = g;
*(*(h + 1) + j) = i - k;

/* Berechnung der Differenz benachbarter Primzahlen */

for (i = 1; i <= n; i++) {
    dp = *(p + i) - *(p + i - 1);
    (*(d + dp))++;
    if (dp > dmax) {
        dmax = dp;
        d1 = *(p + i);
        d2 = *(p + i - 1);
    }
}

time(&zeit2);

/* achtspaltige Ausgabe auf stdout */

printf("\tPrimzahlen bis %lu\n\n", ende);

j = n - ( r = ((n + 1) % 8));
q = p;

for (i = 0; i <= j; i += 8) {
    printf("\t%6lu\t%6lu\t%6lu\t%6lu\t%6lu\t%6lu\t%6lu \
            \t%6lu\n", *q, *(q+1), *(q+2), *(q+3), \
            *(q+4), *(q+5), *(q+6), *(q+7));
q += 8;
}

if (r != 0) {
    printf("\t");
    for (i = 0; i < r; i++)      /* letzte Zeile */
    printf("%6lu\t", *(q+i));
    puts("");
}

printf("\n\tGesamtzahl: %u\n\n", n + 1);

for (i = 1; i <= 10; i++)
    printf("\tZwischen %6lu und %6lu gibt es
    %6u Primzahlen.\n",*(*h+i-1),*(*h+i),*(*(h+1)+i) );

puts("");

printf("\tDifferenz %3d kommt %6u mal vor.\n",\
                                 1, *(d + 1));

for (i = 2; i <= dmax; i += 2)
    printf("\tDifferenz %3d kommt %6u mal vor.\n", \
                                     i, *(d + i));

printf("\n\tGroesste Differenz %lu kommt erstmals
bei %lu und %lu vor.\n",dmax,d2,d1);

time(&zeit3);

printf("\n\tDie Rechnung brauchte %ld s,", zeit2 - zeit1);
printf(" die Ausgabe %ld s.\n", zeit3 - zeit2);
```

```
return 0;
}

/* Ende Hauptprogramm */

/* Funktion zum Testen der Teilbarkeit */
/* Parameteruebergabe zwecks Zeitersparnis vermieden */

void ttest()
{
register int i;

for (i = 1; *(p + i) * *(p + i) <= z; i++)
    if (!(z % *(p + i))) return;        /* z teilbar */
*(p + (++n)) = z;              /* z prim */
return;
}
```

Quelle 1.89 : C-Programm zur Berechnung von Primzahlen, mit Pointern anstelle von Arrayindizes

Zur Laufzeit zeigt sich, daß die meiste Zeit auf die Ausgabe verwendet wird. Daher die Programmiererweisheit: Eingabe/Ausgabe vermeiden! Am Algorithmus und seiner Verwirklichung etwas zu optimieren, bringt für die Gesamtdauer praktisch nichts. Die Ausgabe-Funktion `printf(3)` ließe sich durch eine selbstgeschriebene, schnellere Funktion ersetzenr, unter Abstrichen an die Allgemeingültigkeit.

1.11.7.4 Arrays von Funktionspointern

Der Name einer Funktion ohne das Klammernpaar ist der Pointer auf ihren Anfang. Es gibt Arrays von Pointern, das ist nichts Besonderes. Also gibt es auch Arrays von Pointern auf Funktionen. Anhand eines Beispiels wollen wir uns eine Verwendungsmöglichkeit und einige syntaktische Feinheiten ansehen. Dabei kommt auch `make(1)` nochmal zur Geltung sowie ein bißchen Grafikprogrammierung.

```
/* schiff.c, Befeuerung und Schallsignale nach BinSchStrO
   MS-Quick-C, DOS, VGA-Grafik */

/* #define  DEBUG */

#include <stdio.h>      /* fuer puts() u. a. */
#include <conio.h>      /* fuer getch() und kbhit() */
#include <graph.h>      /* fuer Grafik */
#include <stdlib.h>     /* fuer rand(), Zufallszahlen */
#include <time.h>       /* fuer time(), Zufallszahlen */

#include "schiff.h"     /* eigenes Includefile */

/* Funktionsprototypen */

extern int titel();     /* Titelbildschirm */
void done();            /* Aufraeumen */

/* Hauptprogramm */

int main()
{

int i, rf, x;
struct videoconfig vc;
int (*schiff[MAX])(), index[MAXARR];
time_t zeit;

/* Titel und Vorspann */
```

```
rf = titel();

/* Grafikmodus VGA 16 Farben */

if (!_setvideomode(_VRES16COLOR)) {
        puts("Fehler: Keine VGA-Grafik");
        exit(1);
}
_setbkcolor(BGRAU);

/* Array der Funktionen (Bilder) fuellen */

schiff[0] = bild000; schiff[1] = bild001;
schiff[2] = bild002; schiff[3] = bild003;
....
schiff[124] = bild124; schiff[125] = bild125;

/* Index-Array fuellen (Zufallszahlen) */

if (rf == 49) {
        time(&zeit);
        srand((unsigned)zeit);

        for (i = 0; i < MAXARR; i++) {
                x = rand() % MAX;
                if (x == index[i - 1])
                        x = rand() % MAX;
                index[i] = x;
        }
}
else {
        for (i = 0; i < MAXARR; i++)
                index[i] = i % MAX;
}

#ifdef DEBUG
for (i = 0; i < MAXARR; i++)
        printf("%d\n", index[i]);
while (!kbhit());
#endif

/* Koordinatensystem */
/* x nach rechts, x = 0 in Mitte;
   y nach unten, y = 0 oben */

_getvideoconfig(&vc);
_setlogorg(vc.numxpixels / 2 - 1, 0);

/* Textfenster */

_settextwindow(7 * vc.numtextrows / 8, 4, vc.numtextrows, \
                                        vc.numtextcols - 3);
_wrapon(_GWRAPOFF);

/* Hauptschleife */

i = 0;
do {

/* Hintergrund (Tag - Nacht) */

if (_getbkcolor() == BSCHWARZ) {
        _setbkcolor(BGRAU);
        _setcolor(HWEISS);
}
```

```
else {
        _setbkcolor(BSCHWARZ);
        _setcolor(SCHWARZ);
}

if (_getbkcolor())
        _settextcolor(HWEISS);
else
        _settextcolor(SCHWARZ);

/* Aufruf der Fkt. zum Zeichnen der Schiffe */

(*schiff[index[i]])();

/* Tastatureingabe zum Weitermachen */

if ((x = getch()) == 13 || x == 43 || x == 45) {
        if (x == 45) i-;
        else i++;
        if (i >= MAXARR || i <= 0) i = 0;
        _clearscreen(_GCLEARSCREEN);
        _setbkcolor(BGRAU);
}

} while (x != 81 && x != 113);

done();             /* Ende main() */
}

/* Funktion done() zur Beendigung */
/* Ruecksetzen auf Defaultwerte */

void done()
{
_clearscreen(_GCLEARSCREEN);
_setvideomode(_DEFAULTMODE);
exit(0);
}
```

Quelle 1.90 : C-Programm Array von Pointern auf Funktionen

Das Programm zeichnet die Konturen von Binnenschiffen samt ihrer Befeuerung (Positionslichter etc.) auf den Schirm. Tuten kann es auch. Ist der Hintergrund schwarz (Nacht), sieht man nur die Lichter, ist er grau (Tag), sieht man auch die Konturen und die Bildunterschrift. Die verschiedenen Arten der Befeuerung werden durch jeweils eine Funktion `bild***()` erzeugt. Die Funktionspointer stehen in einem Array:

```
int (*schiff[MAX]);      /* MAX = 126, in schiff.h */
```

Die Funktionen werden über ihren Index aufgerufen, die Reihenfolge wird von einem weiteren Array:

```
int index[MAXARR];       /* MAXARR ein Mehrfaches von MAX */
```

bestimmt, das entweder mit einer wiederholten Folge der natürlichen Zahlen von 0 bis `MAX - 1` belegt ist oder mit einer Zufallsfolge von Zahlen dieses Intervalles. So kann man sich die Befeuerungen in einer systematischen oder zufälligen Folge anzeigen lassen.

Die Funktionen `bild***()` sind in einer Datei in einem eigenen Unterverzeichnis vereinigt:

```
/* Funktionen bild*() */

#include "bilder.h"
```

```
/*************/
/* Sportboot */
/*************/

...

int bild001()
{
text0("Bild 001:");
text1("Sportboot von vorn");
sportboot(0);
feuersport1();
return 0;
}

...
```

Quelle 1.91 : C-Funktion bilder.c zum Programm schiff.c

Die Funktionen rufen im wesentlichen weitere Funktionen auf. Das Unterverzeichnis enthält eine eigene Include-Datei und ein eigenes Makefile. In gleicher Weise sind die übrigen Funktionen organisiert.

Das Makefile des Hauptprogramms ruft Makefiles in den Unterverzeichnissen auf. Wir haben also eine Hierarchie von Makefiles:

```
# makefile fuer schiff.c

include make.h       # Compiler-Auswahl, make-include

# Unterverzeichnisse

A = assem
B = bilder
F = feuer
K = kontur
S = schall
T = text

# Weitere Makros

OBJS = schiff.obj titel.obj bilder.obj text.obj \
       feuer.obj schall.obj kontur.obj sound.obj \
                           nosound.obj delay.obj

# Anweisungen

all : schiff.exe install clean

schiff.exe : schiff.obj titel.obj bilder_o text_o \
                feuer_o kontur_o schall_o assem_o
        $(LD) $(LDFLAGS) $(OBJS)„„

schiff.obj : schiff.c schiff.h
        $(CC) $(CFLAGS) schiff.c

titel.obj : titel.c
        $(CC) $(CFLAGS) titel.c

bilder_o :
        cd $(B)
        $(MAKE) all
        cd ..

text_o :
        cd $(T)
```

```
        $(MAKE) all
        cd ..

feuer_o :
        cd $(F)
        $(MAKE) all
        cd ..

kontur_o :
        cd $(K)
        $(MAKE) all
        cd ..

schall_o :
        cd $(S)
        $(MAKE) all
        cd ..

assem_o  :
        cd $(A)
        $(MAKE) all
        cd ..

install :
        $(CP) schiff.exe s.exe

clean :
        $(RM) *.bak
        $(RM) *.obj
```

Quelle 1.92 : Makefile zu schiff.c

Dieses Projekt – obwohl bescheiden – wäre ohne `make(1)` nur noch mühsam zu beherrschen. Es ist für das Gelingen entscheidend, sich zu Beginn die Struktur sorgfältig zu überlegen.

Infolge der Verwendung von Grafikfunktionen des MS-Quick-C-Compilers ist das Programm nicht auf andere Systeme übertragbar. Man müßte eigene Grafikfunktionen verwenden, die Verpackungen um die Grafikfunktionen des jeweiligen Compilers darstellen. Dasselbe gilt für die Funktion zum Tuten, eine Assemblerroutine. Vielleicht stellen wir das Programm einmal auf X11 um und verpacken dabei die spezifischen Funktionen.

1.11.8 Verarbeitung von Strings

Strings (Zeichenketten) sind in C/C++ ein denkbar einfacher Datentyp, nämlich Arrays of char, die mit dem ASCII-Zeichen Nr. 0 abgeschlossen sind. In der Standard-C-Bibliothek finden sich fertige Funktionen für die häufigsten Aufgaben der Stringverarbeitung. Trotzdem hat dieses Gebiet ein paar kleine Tücken.

Ein beliebter Laufzeitfehler ist das Schreiben von Strings in Pufferspeicher, die zu klein sind. Es kommt dann zu einem Überlauf mit nicht immer vorhersehbaren Folgen, manchmal auch zu von bösen Buben beabsichtigten Folgen. Man meint, der Puffer sei reichlich bemessen – es ist auch jahrelang alles gut gegangen – und dann taucht einmal ein überlanger String auf, der den Puffer sprengt. Das Gegenmittel besteht im Beschränken des Schreibens auf maximal ein Zeichen weniger als der Puffer fasst. Wo Eingaben verlangt werden, ist immer damit zu rechnen, dass:

- die Eingabe leer ist (leerer String, nur ASCII-Nr. 0) oder
- zu lang ist (Überlauf)

Ein gewiefter Programmierer sieht solche Fehler oder Ausnahmen voraus.

gets() – fgets()

1.11.9 Dynamische Speicherverwaltung (malloc)

Wir haben gelernt, daß die Größe eines Arrays oder einer Struktur bereits zur Übersetzungszeit bekannt sein, d. h. im Programm stehen muß. Dies führt in manchen Fällen zur Verschwendung von Speicher, da man Arrays in der maximal möglichen Größe anlegen müßte. Die Standardfunktion `malloc(3)` samt Verwandtschaft hilft aus der Klemme. Im folgenden Beispiel wird ein Array zunächst nur als Pointer `la` deklariert, dann mittels `calloc(3)` Speicher zugewiesen, mittels `realloc(3)` vergrößert und schließlich von `free(3)` wieder freigegeben:

```
/* Programm allo.c zum Ueben von malloc(3), 1994-06-01 */

#define MAX 40
#define DELTA 2

#include <stdio.h>
#include <stdlib.h>

long *la;                    /* Pointer auf long */

int main()
{
int i, x;

/* calloc() belegt Speicher fuer Array von MAX Elementen
   der Groesse sizeof(long), initialisert mit 0,
   gibt Anfangsadresse zurueck.
   In stdlib.h wird size_t als unsigned int definiert. */

la = (long *)calloc((size_t)MAX, (size_t)sizeof(long));

if (la != NULL)
        puts("Zuordnung ok.");
else {
        puts("Ging daneben.");
        exit(-1);
}

/* Array anschauen */

printf("Ganzzahl eingeben: ");
scanf("%d", &x);

for (i = 0; i < MAX; i++)
        la[i] = (long)(i * x);

printf("Ausgabe: %ld    %ld\n", la[10], la[20]);

/* Array verlaengern mit realloc() */

la = (long *)realloc((void *)la, \
     (size_t)(DELTA * sizeof(long)));

/* Array anschauen */

la[MAX + DELTA] = x;

printf("erweitert: %ld    %ld\n", la[10], la[MAX + DELTA]);

/* Speicher freigeben mit free() */

free((void *)la);

return 0;
}
```

Quelle 1.93 : C-Programm mit dynamischer Speicherverwaltung (malloc(3))

Das nächste Beispiel sortiert die Zeilen eines Textes nach den Regeln des Duden (Duden-Taschenbuch Nr. 5: Satz- und Korrekturanweisungen), die von den Regeln in DIN 5007 etwas abweichen.

```
/* "duden" sortiert Textfile zeilenweise nach dem ersten
   Wort unter Beruecksichtigung der Duden-Regeln */
/* Falls das Wort mit einem Komma endet, wird auch das
   naechste Wort beruecksichtigt (z. B. Vorname) */
/* Compiler: cc -O -o duden duden.c -lmalloc */

#include <stdio.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAX 1024              /* max. Anzahl der Zeilen */
#define EXT ".s"              /* Kennung des sort. Files */
#define NOWHITE(c) \
        (((c) != ' ') && ((c) != '\t') && ((c) != '\n'))
#define NOCHAR(c) \
        (((c) == ' ') || ((c) == '\t') || ((c) == '\0'))
#define SCHARF(c) (((c) == '~') || ((c) == 222)) /* sz */
#define KOMMA(c)  ((c) == ',')

/* statische Initialisierung eines externen Arrays */
/* ASCII-Tafel. Die Zahlen stellen die Nummer des
   Zeichens dar. */
/* angefuegt HP ROMAN EXTENSION (optional) */

char wert[256] = {
    /* Steuerzeichen */
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
    20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
    30, 31,
    /* Space, Sonder- und Satzzeichen */
    32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47,
    /* Ziffern */
    65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
    /* Sonder- und Satzzeichen */
    48, 49, 50, 51, 52, 53, 89,
    /* Grossbuchstaben */
    75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
    88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
    /* Sonder- und Satzzeichen */
    75, 89, 95, 58, 59, 60,
    /* Kleinbuchstaben */
    75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
    88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
    /* Sonder- und Satzzeichen */
    75, 89, 95, 93,
    /* DEL */
    111,

    /* ROMAN EXTENSION */
    /* undefinierte Zeichen */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0,
    /* Buchstaben */
    75, 75, 79, 79, 79, 83, 83,
```

```
    /* Zeichen */
    0, 0, 0, 0, 0,
    /* Buchstaben */
    93, 93,
    /* Zeichen */
    0, 0, 0, 0, 0,
    /* Buchstaben */
    77, 77, 88, 88,
    /* Zeichen */
    0, 0, 0, 0, 0, 0, 0, 0,
    /* Buchstaben */
    75, 79, 89, 95, 75, 79, 89, 95,
    75, 79, 89, 95, 75, 79, 89, 95,
    75, 83, 89, 75, 75, 83, 89, 75,
    75, 83, 89, 95, 79, 83, 93, 89,
    75, 75, 75, 78, 78, 83, 83, 89,
    89, 89, 89, 93, 93, 95, 99, 99,
    101, 101,
    /* Zeichen und undefinierte Zeichen */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0
};

char *ap[MAX];              /* P. auf Zeilenanfaenge */

/* Hauptprogramm */

int main(int argc, char *argv[])

{
int flag = 0, i = 0, j;
char a, *mp;
FILE *fp, *fps;
struct stat buf;
extern char *ap[];
extern char *strcat();
void exit();

/* Pruefung des Programmaufrufs */

if (argc != 2) {
    printf("Aufruf: duden FILENAME\n");
    exit(1);
}

/* Arbeitsspeicher allokieren */

stat(argv[1], &buf);
if ((mp = malloc((unsigned)buf.st_size)) == NULL) {
    printf("Kein Speicher frei.\n");
    exit(1);
}
ap[0] = mp;

/* Textfile einlesen, fuehrende NOCHARs loeschen */

if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("File %s kann nicht goeffnet werden.\n", \
                                        argv[1]);
    exit(1);
}

while((a = fgetc(fp)) != EOF) {
    if ((flag == 0) && NOCHAR(a));
    else {
        flag = 1;
```

```
        *mp = a;
        if (*mp == '\n') {
            flag = 0;
            ap[++i] = ++mp;
        }
        else
            mp++;
    }
}

fclose(fp);

/* Zeilenpointer sortieren */

if (sort(i - 1) != 0) {
    printf("Sortieren ging daneben.\n");
    exit(1);
}

/* Textfile zurueckschreiben */

if ((fps = fopen(strcat(argv[1], EXT), "w")) == NULL) {
    printf("File %s.s kann nicht geoeffnet werden.\n", \
                                         argv[1]);
    exit(1);
}

for (j = 0; j < i; j++) {
    while ((a = *((ap[j])++)) != '\n')
        fputc(a, fps);
    fputc('\n', fps);
}

fclose(fps);
}

/* Ende Hauptprogramm */

/* Sortierfunktion (Bubblesort, stabil) */

int sort(int imax)

{
int flag = 0, i = 0, j = 0, k = 0;
char *p1, *p2;
extern char *ap[];

while (flag == 0) {
    flag = 1;
    k = i;
    p2 = ap[imax];
    for (j = imax; j > k; j-) {
        p1 = ap[j - 1];
        if (vergleich(p1, p2) <= 0) {
            ap[j] = p2;
            p2 = p1;
        }
        else {
            ap[j] = p1;
            i = j;
            flag = 0;
        }
    }
    ap[j] = p2;
}
return(0);
```

```
}

/* Vergleich zweier Strings bis zum ersten Whitespace */
/* Returnwert = 0, falls Strings gleich
   Returnwert < 0, falls String1 < String2
   Returnwert > 0, falls String1 > String2 */

int vergleich(char *x1, *x2)

{
int  flag = 0;

while((wert[*x1] - wert[*x2]) == 0) {
    if (NOWHITE(*x1)) {
        if (SCHARF(*x1)) x2++;      /* scharfes s */
        if (SCHARF(*x2)) {
            x1++;
            flag = 1;
        }
        x1++;
        x2++;
    }
    else {
        if (KOMMA(*(x1 - 1))) {     /* weiteres Wort */
            while (NOCHAR(*x1))
                x1++;
            while (NOCHAR(*x2))
                x2++;
            flag = vergleich(x1, x2);
        }
        return flag;
    }
}
return(wert[*x1] - wert[*x2]);
}
```

Quelle 1.94 : C-Programm zum Sortieren eines Textes nach den Regeln des Duden

Die Variable `flag`, die auch anders heißen kann, ist ein **Flag** oder eine Schaltvariable, d. h. eine Variable, die in Abhängigkeit von bestimmten Bedingungen einen Wert 0 oder nicht-0 annimmt und ihrerseits wieder in anderen Bedingungen auftritt. Ein gängiger, einwandfreier Programmiertrick.

1.11.10 X Window System

Das folgende Beispiel zeigt, wie man unter Benutzung von `Xlib`-**Funktionen** ein Programm schreibt, das unter dem X Window System läuft:

```
/* xwindows.c, this program demonstrates how to use
   X's base window system through the Xlib interface */
/* Compiler: cc xwindows.c -lX11 */

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define QUIT   "Press q to quit"
#define CLEAR  "Press c to clear this window"
#define DELETE "Press d to delete this window"
#define SUBWIN "Press n to create subwindow"
#define DELSUB "Press n again to delete window"
#define WIN1   "WINDOW 1"
#define WIN2   "WINDOW 2"
#define WIN3   "WINDOW 3"
```

```
char hallo[]="Hallo World";
char hi[]   ="Hi";

int main(int argc,char **argv)
{
  Display      *mydisplay;              /* d. structure */
  Window       mywin1, mywin2, newwin;  /* w. structure */
  Pixmap       mypixmap;                /* pixmap */
  GC           mygc1, mygc12, newgc;    /* graphic context */
  XEvent       myevent;                 /* event to send */
  KeySym       mykey;                   /* keyboard key */
  XSizeHints   myhint;                  /* window info */
  Colormap     cmap;                    /* color map */
  XColor       yellow, exact, color1, color2, color3;
  static XSegment segments[]={{350,100,380,280}, /
                             {380,280,450,300}};
  unsigned long myforeground;           /* fg color */
  unsigned long mybackground;           /* bg color */
  int          myscreen, i, num=2, del=1, win=1;
  char         text[10];

  /* initialization */
  if (!(mydisplay = XOpenDisplay(""))) {
    fprintf(stderr, "Cannot initiate a display connection");
    exit(1);
  }
  myscreen = DefaultScreen(mydisplay); /* display screen */

  /* default pixel values */
  mybackground = WhitePixel(mydisplay, myscreen);
  myforeground = BlackPixel(mydisplay, myscreen);

  /* specification of window position and size */
  myhint.x = 200; myhint.y = 300;
  myhint.width = 550; myhint.height = 450;
  myhint.flags = PPosition | PSize;

  /* window creation */
  mywin1 = XCreateSimpleWindow(mydisplay,
           DefaultRootWindow(mydisplay),
           myhint.x, myhint.y, myhint.width, myhint.height,
           5, myforeground, mybackground);
  XSetStandardProperties(mydisplay, mywin1, hallo, hallo, \
                            None, argv, argc, &myhint);

  myhint.x = 400; myhint.y = 400;
  myhint.width = 700; myhint.height = 200;
  myhint.flags = PPosition | PSize;
  mywin2 = XCreateSimpleWindow(mydisplay,
           DefaultRootWindow(mydisplay),
           myhint.x, myhint.y, myhint.width, \
           myhint.height, 5, myforeground, mybackground);

  /* creation of a new window */
  XSetStandardProperties(mydisplay, mywin2, "Hallo", \
                 "Hallo", None, argv, argc, &myhint);

  /* pixmap creation */
  mypixmap = XCreatePixmap(mydisplay,
             DefaultRootWindow(mydisplay),
             400, 200, DefaultDepth(mydisplay, myscreen));

  /* GC creation and initialization */
  mygc1 = XCreateGC(mydisplay, mywin1, 0, 0);
  mygc12 = XCreateGC(mydisplay, mywin2, 0, 0);
```

```
newgc = XCreateGC(mydisplay, mywin2, 0, 0);

/* determination of default color map for a screen */
cmap = DefaultColormap(mydisplay, myscreen);
yellow.red = 65535; yellow.green = 65535; yellow.blue = 0;

/* allocation of a color cell */
if (XAllocColor(mydisplay, cmap, &yellow) == 0) {
  fprintf(stderr, "Cannot specify color");
  exit(2);
}

/* allocation of color cell using predefined
   color-name */
if (XAllocNamedColor(mydisplay, cmap, "red", &exact, \
                                      &color1) == 0)
{
  fprintf(stderr, "Cannot use predefined color");
  exit(3);
}
if (XAllocNamedColor(mydisplay, cmap, "blue", &exact, \
                                       &color2) == 0)
{
  fprintf(stderr, "Cannot use predefined color");
  exit(3);
}
if (XAllocNamedColor(mydisplay, cmap, "green", \
                        &exact, &color3) == 0)
{
  fprintf(stderr, "Cannot use predefined color");
  exit(3);
}
XSetWindowBackground(mydisplay, mywin1, color2.pixel);
      /* changing the background of window */
XSetWindowBackground(mydisplay, mywin2, color3.pixel);
XSetBackground(mydisplay, mygc1, color2.pixel);
      /* setting foreground attribute in GC structure */
XSetForeground(mydisplay, mygc1, yellow.pixel);
      /* setting background attribute in GC structure */
XSetForeground(mydisplay, mygc12, color1.pixel);
XSetBackground(mydisplay, mygc12, color3.pixel);
XSetBackground(mydisplay, newgc, mybackground);
XSetFont(mydisplay, mygc1, XLoadFont(mydisplay, \
                                    "vrb-25"));
      /* setting font attribute in GC structure */
XSetFont(mydisplay, mygc12, XLoadFont(mydisplay, \
                                     "vri-25"));
XSetFont(mydisplay, newgc, XLoadFont(mydisplay, \
                                    "vri-25"));
/* window mapping */
XMapRaised(mydisplay, mywin1);
XMapRaised(mydisplay, mywin2);

/* input event selection */
XSelectInput(mydisplay, mywin1, \
  KeyPressMask | ExposureMask);
XSelectInput(mydisplay, mywin2, \
  KeyPressMask | ExposureMask |
             ButtonPressMask);

/* main event-reading loop */
while (1) {
  XNextEvent(mydisplay, &myevent); /* read next event */
  switch (myevent.type) {

   /* process keyboard input */
```

```
case KeyPress:
  i = XLookupString(&myevent, text, 10, &mykey, 0);
  if (i == 1 && (text[0] == 'q' | text[0] == 'Q')) {
    XFreeGC(mydisplay, mygc1);
    XFreeGC(mydisplay, mygc12);
    XFreeGC(mydisplay, newgc);
    if (!win) XDestroyWindow(mydisplay, newwin);
    XDestroyWindow(mydisplay, mywin1);
    if (del) XDestroyWindow(mydisplay, mywin2);
    XFreePixmap(mydisplay, mypixmap);
    XCloseDisplay(mydisplay);
    exit(0);
  }
  else
   if (i == 1 && (text[0] == 'c' | text[0] == 'C') &&
       myevent.xkey.window == mywin1) {
     XClearWindow(mydisplay, mywin1);
     XSetFont(mydisplay, mygc1, \
              XLoadFont(mydisplay, "fgb-13"));
     XDrawImageString(mydisplay, mywin1, mygc1, \
                                        240, 400,
                      SUBWIN, strlen(SUBWIN));
     XDrawImageString(mydisplay, mywin1, mygc1, \
                                        240, 420,
                      CLEAR, strlen(CLEAR));
     XDrawImageString(mydisplay, mywin1, mygc1, \
                                        240, 440,
                      QUIT, strlen(QUIT));
     XSetFont(mydisplay, mygc1, \
              XLoadFont(mydisplay, "vrb-25"));
   }
   else
     if (i == 1 && (text[0] == 'd' | text[0] == 'D') \
                 && myevent.xkey.window == mywin2) {
       XDestroyWindow(mydisplay, mywin2);
       del = 0;
     }
     else
       if (i == 1 && (text[0] == 'n' | text[0] == 'N') \
                   && myevent.xkey.window == mywin1) {
         if (win) {
           newwin = XCreateSimpleWindow(mydisplay, \
                      mywin1, 70, 60, 400, 200, 1, \
                      myforeground, mybackground);
           /* window mapping */
           XMapRaised(mydisplay, newwin);
           XSetForeground(mydisplay, newgc, \
                             mybackground);
           XFillRectangle(mydisplay, mypixmap, newgc,
                          0, 0, 400, 200);
           XSetForeground(mydisplay, newgc, \
                             color1.pixel);
           XDrawImageString(mydisplay, mypixmap, newgc,
                           140, 100, WIN3, strlen(WIN3));
           XSetFont(mydisplay, newgc, \
                    XLoadFont(mydisplay, "fgb-13"));
           XDrawImageString(mydisplay, mypixmap, newgc, 25,
                           180, DELSUB, strlen(DELSUB));
           XSetFont(mydisplay, newgc, \
                    XLoadFont(mydisplay, "vri-25"));
           /* copying pixels from pixmap to window */
           XCopyArea(mydisplay, mypixmap, newwin, newgc, \
                                 0, 0, 400, 200, 0, 0);
         }
         else
           XDestroySubwindows(mydisplay, mywin1);
```

```
              win = !win;
            }
          break;

        /* repaint window on expose event */
        case Expose:
          if (myevent.xexpose.count == 0) {
            XDrawImageString(mydisplay, mywin1, mygc1,
                             50, 50, WIN1, strlen(WIN1));
            XDrawImageString(mydisplay, mywin2, mygc12,
                             270, 50, WIN2, strlen(WIN2));
            XSetFont(mydisplay, mygc1, \
                     XLoadFont(mydisplay, "fgb-13"));
            XDrawImageString(mydisplay, mywin1, mygc1, \
                    240, 400, SUBWIN, strlen(SUBWIN));
            XDrawImageString(mydisplay, mywin1, mygc1, \
                      240, 420, CLEAR, strlen(CLEAR));
            XDrawImageString(mydisplay, mywin1, mygc1, \
                        240, 440, QUIT, strlen(QUIT));
            XSetFont(mydisplay, mygc1, \
                     XLoadFont(mydisplay, "vrb-25"));
            XDrawImageString(mydisplay, mywin2, mygc12, \
                     300, 180, DELETE, strlen(DELETE));
            XDrawLine(mydisplay, mywin1, mygc1, \
                           100, 100, 300, 300);
            XDrawSegments(mydisplay, mywin1, mygc1, \
                                    segments, num);
            XDrawArc(mydisplay, mywin1, mygc1, 200, \
                          160, 200, 200, 0, 23040);
            XFillArc(mydisplay, mywin1, mygc12, 60, \
                          200, 120, 120, 0, 23040);
            XDrawRectangle(mydisplay, mywin1, mygc1, \
                                 60, 200, 120, 120);
          }
          break;

        /* process mouse-button presses */
        case ButtonPress:
          XSetFont(mydisplay, mygc1, \
                   XLoadFont(mydisplay,"vxms-37"));
          XDrawImageString(myevent.xbutton.display, \
                           myevent.xbutton.window, mygc1, \
                           myevent.xbutton.x, \
                           myevent.xbutton.y, \
                           hi, strlen(hi));
          XSetFont(mydisplay, mygc1, \
                   XLoadFont(mydisplay, "vrb-25"));
          break;

        /* process keyboard mapping changes */
        case MappingNotify:
          XRefreshKeyboardMapping(&myevent);
      }
    }
}
```

Quelle 1.95 : C-Programm für das X Window System mit Funktionen der Xlib-Bibliothek

Die Xlib-Biliothek stellt die unterste Stufe der X11-Bibliotheken dar. Nach Möglichkeit verwendet man höhere Bibliotheken, die ihrerseits auf der Xlib aufsetzen. Man erspart sich damit viel Mühe.

1.11.11 cgi-Programme

Die Abkürzung `cgi` bedeutet *Common Gateway Interface*. Das ist ein Protokoll zum Gedankenaustausch zwischen HTML-Forms und Programmen. Die Programme können in jeder Sprache verfasst sein, die es ermöglicht, von `stdin` zu lesen, nach `stdout` zu schreiben und außerdem Umgebungsvariable auszuwerten. Obwohl viele derartige Programme Perl-Skripte sind – siehe Abschnitt **??** *Forms und cgi-Scripts* auf Seite **??** – ist das nicht zwingend, ein C-Programm tut es genau so gut und ist schneller.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!-
  cgi_test.html
  Teil einer Imlpementation der CGI-Schnittstelle in C.
->
<HTML>
  <HEAD>
    <TITLE>
      Test
    </TITLE>
  </HEAD>
  <BODY>
    <FORM action="/cgi-bin/cgi_test.cgi" method="get">
      <!- Method "get" is deprecated ->
      <H2>
        GET-Formular
      </H2>
      <INPUT type="hidden" name="foobar" value="42"><INPUT type=
      "text" name="eingabe"><INPUT type="submit">
    </FORM>
    <HR>
    <FORM action="/cgi-bin/cgi_test.cgi" method="post">
      <H2>
        POST-Formular
      </H2>
      <INPUT type="hidden" name="foobar" value="42"><INPUT type=
      "text" name="eingabe"><INPUT type="submit">
    </FORM>
  </BODY>
</HTML>
```

Quelle 1.96 : Webseite `cgi_test.html` mit `FORM`-Elementen zum Ausprobieren der `cgi`-Schnittstelle

Ein Benutzer ruft mittels seines Brausers eine Webseite auf, die ein `FORM`-Element enthält. Was das im einzelnen ist, tut hier nichts zur Sache und kann in HTML-Anleitungen nachgelesen werden. Unter den Attributen des `FORM`-Elements finden sich `action` und `method`. Die `action` gibt den vollständigen oder relativen URL des Programms an, das auf dem Webserver[36] ausgeführt werden soll. Hinter `method` findet sich entweder `post` oder `get`, wobei letzteres veraltet (deprecated in HTML 4.0) ist.

Der Benutzer gibt in die Webseite verschiedene Werte ein, die durch die `INPUT`-Elemente näher spezifiert sind, und drückt dann ein Knöpfchen, das `submit` oder ähnlich heißt. Darauf hin verlangt der Brauser vom WWW den unter `action` genannten URL unter Beifügung der eingegebenen Werte. Der Webserver stellt fest, dass eines seiner `cgi`-Skripte verlangt wird, und ruft dieses auf.

```
/*
 * cgi.h
 * Teil einer Implementation der CGI-Schnittstelle in C.
 */
```

[36]Etwas völlig anderes sind aktive Inhalte (Anweisungen in Javascript, Java-Applets) von Webseiten, die auf dem Computer des Benutzers ausgeführt werden.

```
#define MAX_QUERY_LEN 4095      /* gegen Missbrauch */

struct list_item {
    struct list_item *next;
    char *key, *value;
};

struct array_item {
    char *key, *value;
};

void unescape_str(char *);      /* eigene Funktionen */
char *get_query();
struct list_item *parse_list(char *);
struct array_item *parse_array(char *);
```

Quelle 1.97 : Include-Datei cgi.h zum Programm cgi_test.c

```
/*
 * cgi_test.c, Hauptprogramm
 * Teil einer Implementation der CGI-Schnittstelle in C.
 * normalerweise entweder linked list oder array verwenden
 */

#include <stdio.h>
#include <string.h>
#include "cgi.h"

int main() {
    char *list_query, *array_query;
    struct list_item *param_list;
    struct array_item *param_array;

    /* hiermit beginnt die Ausgabe */
    printf("Content-Type: text/plain\015\012\015\012");

    /* query string ermitteln und duplizieren */
    list_query = get_query();
    if (list_query == NULL) {
        printf("Keine Parameter!\n");
        return 0;
    }
    array_query = strdup(list_query);

    /* linked list erzeugen */
    param_list = parse_list(list_query);
    printf("Parameter per Linked List:\n");
    while (param_list != NULL) {
        printf("%s = %s\n", param_list->key, param_list->value);
        param_list = param_list->next;
    }

    /* array erzeugen */
    param_array = parse_array(array_query);
    printf("\nParameter per Array:\n");
    while (param_array->key != NULL) {
        printf("%s = %s\n", param_array->key, param_array->value);
        param_array++;
    }

    return 0;
}
```

Quelle 1.98 : C-Programm cgi_test.c zum Ausprobieren der `cgi`-Schnittstelle

```
/*
 * cgi.c, Funktionen
 * Teil einer Implementation der CGI-Schnittstelle in C.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "cgi.h"

/*
 * Wandelt alle %xx-Tripel in einzelne Zeichen um und '+' in ' ',
 * wobei der uebergebene String veraendert wird!
 */
void unescape_str(char *s) {
    size_t i;

    for(i = strlen(s); i > 2; i-, s++) {
        if (*s == '+')
            *s = ' ';
        else if (*s == '%' && isxdigit(s[1]) && isxdigit(s[2])) {
            *s = (isdigit(s[1]) ? s[1] - '0' : tolower(s[1]) - 'a' + 10) « 4 |
                 (isdigit(s[2]) ? s[2] - '0' : tolower(s[2]) - 'a' + 10);
            memmove(s + 1, s + 3, i -= 2);
        }
    }
}

/*
 * Gibt den query string zurueck, wenn keine Fehler auftreten,
 * ansonsten NULL. Unterscheidet dabei zwischen GET und POST.
 * Darf bei POST nur einmal aufgerufen werden!
 */
char *get_query() {
    char *p, *q;
    size_t len;

    /* REQUEST_METHOD suchen */
    p = getenv("REQUEST_METHOD");
    if (p == NULL) return NULL;      /* keine REQUEST_METHOD */

    /* 1. Fall: GET */
    if (strcmp(p, "GET") == 0) {
        p = getenv("QUERY_STRING");
        if (p == NULL) return NULL;  /* kein QUERY_STRING bei GET */
        len = strlen(p);
        if (len < 1 || len > MAX_QUERY_LEN) return NULL;
        q = malloc(len + 1);
        if (q == NULL) return NULL;
        memmove(q, p, len + 1);      /* spart Linken von strcpy */
        return q;

    /* 2. Fall: POST */
    } else if (strcmp(p, "POST") == 0) {
        p = getenv("CONTENT_LENGTH");
        if (p == NULL) return NULL;  /* keine CONTENT_LENGTH */
        len = strtoul(p, NULL, 0);
        if (len < 1 || len > MAX_QUERY_LEN) return NULL;
        q = malloc(len + 1);
        if (q == NULL) return NULL;
        if (fread(q, 1, len, stdin) < len)
            return NULL;    /* Laenge von stdin < CONTENT_LENGTH */
```

```
        q[len] = '\0';
        return q;

    /* 3. Fall: weder noch - kann von diesem Programm
       nicht behandelt werden */
    } else return NULL;
}

/*
 * Gibt eine linked list des geparsten query string zurueck,
 * wenn keine Fehler auftreten, ansonsten NULL. Veraendert
 * den uebergebenen String!
 */
struct list_item *parse_list(char *query) {
    struct list_item first = { NULL }, *last = &first;

    while (*query != '\0') {
        /* neues Listenelement allozieren */
        last->next = malloc(sizeof *last);
        if(last == NULL) return first.next;
        last = last->next;
        last->next = NULL;

        /* einen Parameter extrahieren */
        last->key = query;
        do {
            query++;
        } while (*query != '\0' && (*query != '&' || *(query + 1) == '\0'));
        if (*query != '\0') *query++ = '\0';

        /* in key und value splitten */
        last->value = last->key;
        do {
            last->value++;
        } while (*last->value != '\0' && *last->value != '=');
        if (*last->value != '\0') *(last->value++) = '\0';

        /* "%xx"-Tripel umwandeln */
        unescape_str(last->key);
        unescape_str(last->value);
    }
    return first.next;
}

/*
 * Gibt ein Array von Strukturen zurueck mit den geparsten
 * Parametern, falls keine Fehler auftreten, ansonsten NULL.
 * Geht dabei zweimal ueber den query string, was aber nicht
 * weiter schlimm ist. Das Array wird durch eine Struktur
 * abgeschlossen, bei der beide Werte (key und value) NULL sind.
 * Veraendert den uebergebenen String!
 */
struct array_item *parse_array(char *query) {
    int count;
    char *temp;
    struct array_item *first, *last;

    /* Anzahl Parameter bestimmen */
    for (count = 0, temp = query; *temp != '\0'; count++) {
        do {
            temp++;
        } while (*temp != '\0' && (*temp != '&' || *(temp + 1) == '\0'));
    }
    if (count == 0) return NULL;

    /* Array allozieren */
```

```
    last = first = malloc(sizeof *first * (count + 1));
    if (last == NULL) return NULL;
    first[count].key = first[count].value = NULL;

    while (*query != '\0') {
        /* einen Parameter extrahieren */
        last->key = query;
        do {
            query++;
        } while (*query != '\0' && (*query != '&' || *(query + 1) == '\0'));
        if (*query != '\0') *query++ = '\0';

        /* in key und value splitten */
        last->value = last->key;
        do {
            last->value++;
        } while (*last->value != '\0' && *last->value != '=');
        if (*last->value != '\0') *(last->value++) = '\0';

        /* "%xx"-Tripel umwandeln */
        unescape_str(last->key);
        unescape_str(last->value);

        last++;
    }
    return first;
}
```

Quelle 1.99 : C-Funktionen zum Programm cgi_test.c

Nun wird es spannend. Wie erfährt das `cgi`-Skript, hier ein C-Programm, von den vom Benutzer eingegebenen Werten? Bei der Methode `get` stehen die Werte als ein langer String in der Umgebungs-Variablen `QUERY_STRING`, bei der Methode `post` wird der String von `stdin` gelesen. Der Rest besteht im Aufdröseln (parsen) des Strings, der Verarbeitung der einzelnen Werte und der Ausgabe eines HTML-konformen Dokumentes nach `stdout`. Dieses Dokument schickt der Webserver an den anfragenden Brauser zurück. Das wars. Fehler macht man vor allem bei den Kleinigkeiten wie Gänsefüßchen oder Semikolons. Ein großer Teil des Programms besteht wie üblich im Abfangen unzulässiger Eingaben. Bedenken Sie, dass wildfremde Benutzer dieses Programm auf Ihrem WWW-Server starten. Lesen Sie das Programmbeispiel und weitere aus dem Netz. Dort finden sich auch Funktions- und Klassenbibliotheken für `cgi`-Programme in C/C++.

1.12 Obfuscated C

Wie bereits in einer Fußnote auf Seite 9 bemerkt, findet jährlich ein Wettbewerb um das undurchsichtigste C-Programm statt (to obfuscate = vernebeln, verwirren). Die Siegerprogramme haben außer Nebel auch noch einen Witz aufzuweisen. Als Beispiel geben wir ein Programm von JACK APPLIN, Hewlett-Packard, Fort Collins/USA wieder, das erfolgreich am Contest 1986 teilgenommen hat. Es ist das Hello-World-Programm in einer Fassung, die als C-Programm, FORTRAN-77-Programm und als Bourne-Shellscript gültig ist[37]

```
cat =13 /*/ >/dev/null 2>&1; echo "Hello, world!"; exit
*
*  This program works under cc, f77, and /bin/sh.
*
*/; main() {
      write(
cat-~-cat
```

[37]Unter `www.ee.ryerson.ca:8080/~elf/hack/multilang.htm` liegt ein ähnliches Programm, das als C-Programm, Perl-Skript, Tcl-Skript und Shell-Skript gültig ist.

```
      /*,'(
*/
      ,"Hello, world!"
      ,
cat); putchar(~-~-~-cat); } /*
      ,)')
      end
*/
```

Auch die Leerzeichen sind wichtig. Entfernt man die Kommentare, bleibt als C-Programm übrig:

```
cat =13;
main() {
write(cat-~-cat, "Hello, world!", cat);
putchar(~-~-~-cat); }
```

Zuerst wird eine globale Variable `cat` – per Default vom Typ `int` – auf 13 gesetzt. Dann wird der Systemaufruf `write(1, "Hello, world!", 13)` ausgeführt, der 13 Zeichen des Strings `Hello, world!` nach `stdout` (Dateideskriptor 1) schreibt, anschließend die Standardfunktion `putchar(10)`. Der Gebrauch des unären Minuszeichens samt der bitweisen Negation ist ungewohnt. Man muß sich die Umrechnungen in Bits aufschreiben (negative Zahlen werden durch ihr Zweierkomplement dargestellt). Bei `write()`:

```
13            gibt 0000 0000  0000 0000  0000 0000  0000 1101
-13           gibt 1111 1111  1111 1111  1111 1111  1111 0011
~(-13)        gibt 0000 0000  0000 0000  0000 0000  0000 1100
13 - (~(-13))
              gibt 0000 0000  0000 0000  0000 0000  0000 0001
```

was dezimal 1 ist. Bei `putchar()` sieht die Geschichte so aus:

```
13            gibt 0000 0000  0000 0000  0000 0000  0000 1101
-13           gibt 1111 1111  1111 1111  1111 1111  1111 0011
~(-13)        gibt 0000 0000  0000 0000  0000 0000  0000 1100
-(~(-13))     gibt 1111 1111  1111 1111  1111 1111  1111 0100
~(-(~(-13)))
              gibt 0000 0000  0000 0000  0000 0000  0000 1011
-(~(-(~(-13))))
              gibt 1111 1111  1111 1111  1111 1111  1111 0101
~(-(~(-(~(-13)))))
              gibt 0000 0000  0000 0000  0000 0000  0000 1010
```

was dezimal 10 = ASCII-Zeichen Linefeed ist.

Für FORTRAN 77 bleiben folgende Zeilen übrig:

```
      write(*, '("Hello, world!")')
      end
```

Zeilen, die an erster Stelle ein c oder ein * enthalten, gelten als Kommentar. Zeilen, die an sechster Stelle irgendein Zeichen enthalten, werden als Fortsetzungen aufgefaßt. Anweisungen beginnen in Spalte 7 (die Sitte stammt aus der Lochkartenzeit).

Das Shellscript enthält als einzige wirksame Kommandos:

```
echo "Hello, world!"; exit
```

Was davor steht, geht nach `/dev/null`. Mit `exit` wird das Script verlassen. Mehr solcher Scherze findet man im Netz oder in dem Buch von DON LIBES.

Eine ähnliche intellektuelle Herausforderung stellen die Quine-Programme dar. Das sind Programme, die ihren eigenen Quellcode ausgeben. Näheres unter:

`http://www.nyx.net/~gthompso/quine.htm`

1.13 Portieren von Programmen

1.13.1 Regeln

Unter dem Übertragen oder **Portieren** von Programmen versteht man das Anpassen an ein anderes System unter Beibehaltung der Programmiersprache oder das Übersetzen in eine andere Programmiersprache auf demselben System, schlimmstenfalls beides zugleich.

Ein Programm läßt sich immer portieren, indem man bis zur Aufgabenstellung zurückgeht. Das ist mit dem maximalen Aufwand verbunden; es läuft auf Neuschreiben hinaus. Unter günstigen Umständen kann ein Programm Zeile für Zeile übertragen werden, ohne die Aufgabe und die Algorithmen zu kennen. In diesem Fall reicht die Intelligenz eines Computers zum Portieren; es gibt auch Programme für diese Tätigkeit[38]. Die wirklichen Aufgaben liegen zwischen diesen beiden Grenzfällen.

Schon beim ersten Schreiben eines Programmes erleichtert man ein künftiges Portieren, wenn man einige Regeln beherzigt. Man vermeide:

- Annahmen über Eigenheiten des Datei-Systems (z. B. Länge der Namen),
- Annahmen über die Reihenfolge der Auswertung von Ausdrücken, Funktionsargumenten oder Nebeneffekten (z. B. bei `printf(3)`),
- Annahmen über die Anordnung der Daten im Arbeitsspeicher,
- Annahmen über die Anzahl der signifikanten Zeichen von Namen,
- Annahmen über die automatische Initialisierung von Variablen,
- den Gebrauch von stillschweigenden (automatischen) Typumwandlungen, zum Beispiel von `long` nach `int` unter der Annahme, daß die beiden Typen gleich lang sind,
- das Mischen von vorzeichenlosen und vorzeichenbehafteten Werten,
- die Dereferenzierung von Nullpointern (Null ist keine Adresse),
- Annahmen über die Darstellung von Pointern (Pointer sind keine Ganzzahlen), Zuweisungen von Pointerwerten an `int`- oder `long`-Variable,
- die Annahme, einen Pointer dereferenzieren zu können, der nicht richtig auf eine Datengrenze ausgerichtet ist (Alignment),
- die Annahme, daß Groß- und Kleinbuchstaben unterschieden werden,
- die Annahme, daß der Typ `char` vorzeichenbehaftet oder vorzeichenlos ist (EOF = -1?),
- Bitoperationen mit vorzeichenbehafteten Ganzzahlen,
- die Verwendung von Bitfeldern mit anderen Typen als `unsigned`,
- Annahmen über das Vorzeichen des Divisionsrestes bei der ganzzahligen Division,
- die Annahme, daß eine `extern`-Deklaration in einem Block auch außerhalb des Blockes gilt.

Diese und noch einige Dinge werden von unterschiedlichen Betriebssystemen und Compilern unterschiedlich gehandhabt, und man weiß nie, was einem begegnet. Dagegen soll man:

- den Syntax-Prüfer `lint(1)` befragen,
- Präprozessor-Anweisungen und `typedef` benutzen, um Abhängigkeiten einzugrenzen,
- alle Variablen, Pointer und Funktionen ordentlich deklarieren,
- Funktions-Prototypen verwenden,
- symbolische Konstanten (`#define`) anstelle von rätselhaften Werten im Programm verwenden,

[38]Im GNU-Projekt finden sich ein Program `f2c` (lies: f to c) zum Übertragen von FORTRAN nach C und ein Programm `p2c` zum Portieren von PASCAL nach C.

- richtig ausgerichtete Unions anstelle von trickreichen Überlagerungen von Typen verwenden,
- den `sizeof()`-Operator verwenden, wenn man die Größe von Typen oder Variablen braucht,
- daran denken, daß die Größe von Datentypen je nach Architektur unterschiedlich ist,
- umfangreiche Deklarationen in Include-Dateien packen,
- nur die C-Standard-Funktionen verwenden oder für andere Funktionen die Herkunft oder den Quellcode angeben, mindestens aber die Funktionalität und die Syntax,
- bei `printf(3)` oder `scanf(3)` die richtigen Platzhalter verwenden (`%ld` für `long`),
- alle unvermeidlichen Systemabhängigkeiten auf wenige Stellen konzentrieren und deutlich kommentieren.

Im folgenden wollen wir einige Beispiele betrachten, die nicht allzu lang und daher auch nur einfach sein können.

1.13.2 Übertragen von ALGOL nach C

Wir haben hier ein ALGOL-Programm von RICHARD WAGNER aus dem Buch von KARL NICKEL *ALGOL-Praktikum* (1964) ausgewählt, weil es mit Sicherheit nicht im Hinblick auf eine Übertragung nach C geschrieben worden ist. Es geht um die Bestimmung des größten gemeinsamen Teilers mit dem Algorithmus von EUKLID. Daß wir die Aufgabe und den Algorithmus kennen, erleichtert die Arbeit, daß außer einigen Graubärten niemand mehr ALGOL kennt, erschwert sie.

```
'BEGIN' 'COMMENT' BEISPIEL 12 ;
'INTEGER' A, B, X, Y, R ;
L1:
READ(A,B) ;
'IF' A 'NOT LESS' B
'THEN''BEGIN' X:= A ; Y:= B 'END'
'ELSE''BEGIN' X:= B ; Y:= A 'END' ;
L2:
R:= X - Y*ENTIER(X/Y) ;
'IF' R 'NOT EQUAL' 0 'THEN''BEGIN' X:= Y ; Y:= R ;
'GO TO' L2 'END' ;
PRINT(A,B,Y) ;
'GO TO' L1
'END'
```

Quelle 1.100 : ALGOL-Programm ggT nach Euklid

Die Einlese- und Übersetzungszeit auf einer Z22 betrug 50 s, die Rechen- und Druckzeit 39 s. Damals hatten schnelle Kopfrechner noch eine Chance. Eine Analyse des Quelltextes ergibt:

- Das Programm besteht aus *einer* Datei mit dem Hauptprogramm (war kaum anders möglich),
- Schlüsselwörter stehen in Hochkommas,
- logische Blöcke werden durch `begin` und `end` begrenzt,
- es kommen nur ganzzahlige Variable vor,
- es wird Ganzzahl-Arithmetik verwendet,
- an Funktionen treten `read()` und `print()` auf,
- an Kontrollanweisungen werden `if - then - else` und `goto` verwendet.

Das sieht hoffnungsvoll aus. Die Übertragung nach C:

```
/* Groesster gemeinsamer Teiler nach Euklid
   Uebertragung eines ALGOL-Programms aus K. Nickel nach C
   zu compilieren mit cc -o ggt ggt.c    */

#include <stdio.h>

int main()
{
int a, b, x, y, r;

while(1) {

/* Eingabe */

    puts("ggT von a und b nach Euklid");
        puts("Beenden mit Eingabe 0");
    printf("Bitte a und b eingeben: ");
    scanf("%d %d", &a, &b);

/* Beenden, falls a oder b gleich 0 */

    if ((a == 0) || (b == 0)) exit(0);

/* x muss den groesseren Wert aus a und b enthalten */

    if (a >= b) { x = a; y = b; }
    else        { x = b; y = a; }

/* Euklid */

    while (r = x % y) {
        x = y;
        y = r;
    }

/* Ausgabe */

    printf("%d und %d haben den ggT %d\n", a, b, y);
}
}
```

Quelle 1.101 : C-Programm ggT nach Euklid

Der auch nach UNIX-Maßstäben karge Dialog des ALGOL-Programms wurde etwas angereichert, die `goto`-Schleifen wurden durch `while`-Schleifen ersetzt und der ALGOL-Behelf zur Berechnung des Divisionsrestes (`entier`) durch die in C vorhandene Modulo-Operation.

Bei einem Vergleich mit dem Programm 1.55 *C-Programm ggt nach Euklid, rekursiv* auf Seite 96 sieht man, wie unterschiedlich selbst ein so einfacher Algorithmus programmiert werden kann. Dazu kommen andere Algorithmen zur Lösung derselben Aufgabe, beispielsweise das Ermitteln aller Teiler der beiden Zahlen und das Herausfischen des ggT.

1.13.3 Übertragen von FORTRAN nach C

Gegeben sei ein einfaches Programm zur Lösung quadratischer Gleichungen in FORTRAN77:

```
c      --------------------------------
c      Loesung der quadratischen Gleichung
c      a*x*x + b*x + c = 0
c      reelle Koeffizienten, Loesungen auch komplex
c      --------------------------------
       program quad
c
       real    a,b,c,d,h,r,s,x1,x2
       real    eps
```

```
      complex x1c,x2c
      data    eps/1.0e-30/
c
      write(*,*) 'Loesung von  a*x*x + b*x + c = 0'
      write(*,*) 'Bitte  a, b, und c eingeben'
      read (*,*) a,b,c
c     --------------------------------
c     1. Fall :  a nahe Null, lineare Gleichung
c     --------------------------------
      if (abs(a) .lt. eps) then
          write(*,*) 'WARNUNG :  a nahe Null,
     1                       Null angenommen'
          if (abs(b) .lt. eps) then
              write(*,*) 'WARNUNG :  auch b nahe Null,
     1                                         Unsinn'
              goto 100
          else
              write(*,*) 'Loesung :  x = ',-c/b
              goto 100
          endif
      else
c     --------------------------------
c     Berechnung der Diskriminanten d
c     --------------------------------
          d = b*b - 4.0*a*c
          h = a+a
c     --------------------------------
c     2. Fall :  eine oder zwei reelle Loesungen
c     --------------------------------
          if  ( d .ge. 0.0 )  then
              s = sqrt(d)
              x1 = (-b + s) / h
              x2 = (-b - s) / h
              write(*,*) 'Eine oder zwei reelle
     1                                      Loesungen'
              write(*,*) 'x1 = ', x1
              write(*,*) 'x2 = ', x2
              goto 100
c     --------------------------------
c     3. Fall :  konjugiert komplexe Loesungen
c     --------------------------------
          else
              r = -b / h
              s = sqrt(-d) / h
              x1c = cmplx(r,s)
              x2c = cmplx(r,-s )
              write(*,*) 'Konjugiert komplexe
     1                                      Loesungen'
              write(*,*) 'x1 = ', x1c
              write(*,*) 'x2 = ', x2c
              goto 100
          endif
      endif
c     --------------------------------
c     Programmende
c     --------------------------------
100   stop
      end
```

Quelle 1.102 : FORTRAN-Programm Quadratische Gleichung mit reellen Koeffizienten

Eine Analyse des Quelltextes ergibt:

- Das Programm besteht aus *einer* Datei mit *einem* Hauptprogramm,
- es kommen reelle und komplexe Variable vor,

- es wird Gleitkomma-Arithmetik verwendet, aber keine Komplex-Arithmetik (was die Übertragung nach C erleichtert),
- an Funktionen treten abs(), sqrt() und cmplx() auf,
- an Kontrollanweisungen werden `if - then - else - endif` und `goto` verwendet.

Wir werden etwas Arbeit mit den komplexen Operanden haben. Die Sprunganweisung `goto` gibt es zwar in C, aber wir bleiben standhaft und vermeiden sie. Alles übrige sieht einfach aus.

Als Ersatz für den komplexen Datentyp bietet sich ein Array of `float` oder `double` an. Eine Struktur wäre auch möglich. Falls komplexe Arithmetik vorkäme, müßten wir uns die Operationen selbst schaffen. Hier werden aber nur die komplexen Zahlen ausgegeben, was harmlos ist. Das `goto` wird hier nur gebraucht, um nach der Ausgabe der Lösung ans Programmende zu springen. Wir werden in C dafür eine Funktion `done()` aufrufen. Das nach C übertragene Programm:

```
/* Loesung der quadratischen Gleichung a*x*x + b*x + c = 0
   reelle Koeffizienten, Loesungen auch komplex
   zu compilieren mit cc quad.c -lm  */

#define EPS 1.0e-30            /* Typ double! */

#include <stdio.h>             /* wg. puts, printf, scanf */
#include <math.h>              /* wg. fabs, sqrt */

int done();

int main()
{
double a, b, c, d, h, s, x1, x2;
double z[2];

puts("Loesung von a*x*x + b*x + c = 0");
puts("Bitte a, b und c eingeben");
scanf("%lf %lf %lf", &a, &b, &c);

/* 1. Fall: a nahe Null, lineare Gleichung */

if (fabs(a) < EPS) {
    puts("WARNUNG: a nahe Null, als Null angenommen");
    if (fabs(b) < EPS) {
        puts("WARNUNG: auch b nahe Null, Unsinn");
        done();
    }
    else {
        printf("Loesung: %lf\n", -c/b);
        done();
    }
}
else {

/* Berechnung der Diskriminanten d */

    d = b * b - 4.0 * a * c;
    h = a + a;

/* 2. Fall: eine oder zwei reelle Loesungen */

    if (d >= 0.0) {
        s = sqrt(d);
        x1 = (-b + s) / h;
        x2 = (-b - s) / h;
        puts("Eine oder zwei reelle Loesungen");
        printf("x1 = %lf\n", x1);
        printf("x2 = %lf\n", x2);
```

```
        done();
    }
    else {

/* 3. Fall: konjugiert komplexe Loesungen */

        z[0] = -b / h;
        z[1] = sqrt(-d) / h;
        puts("Konjugiert komplexe Loesungen");
        printf("x1 = (%lf   %lf)\n", z[0], z[1]);
        printf("x2 = (%lf  %lf)\n", z[0], -z[1]);
        done();
    }
}
}

/* Funktion done() zur Beendigung des Programms */

int done()
{
return(0);
}
```

Quelle 1.103 : C-Programm Quadratische Gleichung mit reellen Koeffizienten und komplexen Lösungen, aus FORTRAN übertragen

Bei der Übertragung haben wir keinen Gebrauch von unseren Kenntnissen über quadratische Gleichungen gemacht, sondern ziemlich schematisch gearbeitet. Mathematische Kenntnisse sind trotzdem hilfreich, auch sonst im Leben.

Wir erhöhen den Reiz der Aufgabe, indem wir auch komplexe Koeffizienten zulassen: Schließlich wollen wir das Programm als Funktion (Subroutine) schreiben, die von einem übergeordneten Programm aufgerufen wird:

1.14 Exkurs über Algorithmen

Der Begriff **Algorithmus** – benannt nach einem usbekischen Mathematiker des 9. Jahrhunderts – kommt im vorliegenden Text selten vor, taucht aber in fast allen Programmierbüchern auf. Ein beträchtlicher Teil der Informatik befaßt sich damit. Locker ausgedrückt ist ein Algorithmus eine Vorschrift, die mit endlich vielen Schritten zur Lösung eines gegebenen Problems führt. Ein Programm ist die Umsetzung eines Algorithmus in eine Programmiersprache. Algorithmen werden mit Worten, Formeln oder Grafiken dargestellt. Ein Existenzbeweis ist in der Mathematik schon ein Erfolg, in der Technik brauchen wir einen Lösungsweg, einen Algorithmus.

Das klingt alltäglich. Das Rezept zum Backen einer Prinzregententorte[39] oder die Beschreibung des Aufstiegs auf die Hochwilde in den Ötztaler Alpen[40] sind demnach Algorithmen. Einige Anforderungen an Algorithmen sind:

- **Korrektheit**. Das klingt selbstverständlich, ist aber meist schwierig zu beweisen. Und Korrektheit in einem Einzelfall besagt gar nichts. Umgekehrt beweist bereits ein Fehler die Inkorrektheit.
- **Eindeutigkeit**. Das stellt Anforderungen an die Darstellungsweise, die Sprache; denken Sie an eine technische Zeichnung oder an Klaviernoten. Verschiedene Ausführungswege sind zulässig, bei gleichen Eingaben muß das gleiche Ergebnis herauskommen.

[39]Dr. Oetker Backen macht Freude, Ceres-Verlag, Bielefeld. Die Ausführung dieses Algorithmus läßt sich teilweise parallelisieren.

[40]H. KLIER, Alpenvereinsführer Ötztaler Alpen, Bergverlag Rudolf Rother, München. Der Algorithmus muß sequentiell abgeschwitzt werden.

- **Endlichkeit**. Die Beschreibung des Algorithmus muß eine endliche Länge haben, sonst könnte man ihn endlichen Wesen nicht mitteilen. Er muß ferner eine endliche Ausführungszeit haben, man möchte seine Früchte ja noch zu Lebzeiten ernten. Er darf zur Ausführung nur eine endliche Menge von Betriebsmitteln belegen.
- **Allgemeinheit**. $3 \times 4 = 12$ ist kein Algorithmus, wohl aber die Vorschrift, wie man die Multiplikation auf die Addition zurückführt.

Man kann die Anforderungen herabschrauben und kommt dabei zu reizvollen Fragestellungen, aber für den Anfang gilt obiges. Eine fünfte, technisch wie theoretisch bedeutsame Forderung ist die nach einem guten, zweckmäßigen Algorithmus oder gar die nach dem besten. Denken Sie an die vielen Sortierverfahren (keines ist das *beste* für alle Fälle).

Es gibt – sogar ziemlich leicht verständliche – Aufgaben, die nicht mittels eines Algorithmus zu lösen sind. Falls Sie Bedarf an solchen Nüssen haben, suchen Sie unter dem Stichwort *Entscheidbarkeit* in Werken zur Theoretischen Informatik.

... aber die Daten fehlen, um den ganzen
Nonsens richtig zu überblicken –
Benn, Drei alte Männer

A Zahlensysteme

Außer dem **Dezimalsystem** sind das **Dual-**, das **Oktal-** und das **Hexadezimalsystem** gebräuchlich. Ferner spielt das **Binär codierte Dezimalsystem (BCD)** bei manchen Anwendungen eine Rolle. Bei diesem sind die einzelnen Dezimalstellen für sich dual dargestellt. Die folgende Tabelle enthält die Werte von 0 bis dezimal 255. Bequemlichkeitshalber sind auch die zugeordneten ASCII-Zeichen aufgeführt.

dezimal	dual	oktal	hex	BCD	ASCII
0	0	0	0	0	nul
1	1	1	1	1	soh
2	10	2	2	10	stx
3	11	3	3	11	etx
4	100	4	4	100	eot
5	101	5	5	101	enq
6	110	6	6	110	ack
7	111	7	7	111	bel
8	1000	10	8	1000	bs
9	1001	11	9	1001	ht
10	1010	12	a	1.0	lf
11	101	13	b	1.1	vt
12	1100	14	c	1.10	ff
13	1101	15	d	1.11	cr
14	1110	16	e	1.100	so
15	1111	17	f	1.101	si
16	10000	20	10	1.110	dle
17	10001	21	11	1.111	dc1
18	10010	22	12	1.1000	dc2
19	10011	23	13	1.1001	dc3
20	10100	24	14	10.0	dc4
21	10101	25	15	10.1	nak
22	10110	26	16	10.10	syn
23	10111	27	17	10.11	etb
24	11000	30	18	10.100	can
25	11001	31	19	10.101	em
26	11010	32	1a	10.110	sub
27	11011	33	1b	10.111	esc
28	11100	34	1c	10.1000	fs
29	11101	35	1d	10.1001	gs
30	11110	36	1e	11.0	rs
31	11111	37	1f	11.1	us
32	100000	40	20	11.10	space
33	100001	41	21	11.11	!
34	100010	42	22	11.100	”
35	100011	43	23	11.101	#
36	100100	44	24	11.110	$
37	100101	45	25	11.111	%
38	100110	46	26	11.1000	&
39	100111	47	27	11.1001	’

40	101000	50	28	100.0	(
41	101001	51	29	100.1	)
42	101010	52	2a	100.10	*
43	101011	53	2b	100.11	+
44	101100	54	2c	100.100	,
45	101101	55	2d	100.101	-
46	101110	56	2e	100.110	.
47	101111	57	2f	100.111	/
48	110000	60	30	100.1000	0
49	110001	61	31	100.1001	1
50	110010	62	32	101.0	2
51	110011	63	33	101.1	3
52	110100	64	34	101.10	4
53	110101	65	35	101.11	5
54	110110	66	36	101.100	6
55	110111	67	37	101.101	7
56	111000	70	38	101.110	8
57	111001	71	39	101.111	9
58	111010	72	3a	101.1000	:
59	111011	73	3b	101.1001	;
60	111100	74	3c	110.0	<
61	111101	75	3d	110.1	=
62	111110	76	3e	110.10	>
63	111111	77	3f	110.11	?
64	1000000	100	40	110.100	@
65	1000001	101	41	110.101	A
66	1000010	102	42	110.110	B
67	1000011	103	43	110.111	C
68	1000100	104	44	110.1000	D
69	1000101	105	45	110.1001	E
70	1000110	106	46	111.0	F
71	1000111	107	47	111.1	G
72	1001000	110	48	111.10	H
73	1001001	111	49	111.11	I
74	1001010	112	4a	111.100	J
75	1001011	113	4b	111.101	K
76	1001100	114	4c	111.110	L
77	1001101	115	4d	111.111	M
78	1001110	116	4e	111.1000	N
79	1001111	117	4f	111.1001	O
80	1010000	120	50	1000.0	P
81	1010001	121	51	1000.1	Q
82	1010010	122	52	1000.10	R
83	1010011	123	53	1000.11	S
84	1010100	124	54	1000.100	T
85	1010101	125	55	1000.101	U
86	1010110	126	56	1000.110	V
87	1010111	127	57	1000.111	W
88	1011000	130	58	1000.1000	X
89	1011001	131	59	1000.1001	Y
90	1011010	132	5a	1001.0	Z
91	1011011	133	5b	1001.1	[
92	1011100	134	5c	1001.10	\
93	1011101	135	5d	1001.11	]
94	1011110	136	5e	1001.100	^
95	1011111	137	5f	1001.101	_
96	1100000	140	60	1001.110	‘

97	1100001	141	61	1001.111	a
98	1100010	142	62	1001.1000	b
99	1100011	143	63	1001.1001	c
100	1100100	144	64	1.0.0	d
101	1100101	145	65	1.0.1	e
102	1100110	146	66	1.0.10	f
103	1100111	147	67	1.0.11	g
104	1101000	150	68	1.0.100	h
105	1101001	151	69	1.0.101	i
106	1101010	152	6a	1.0.110	j
107	1101011	153	6b	1.0.111	k
108	1101100	154	6c	1.0.1000	l
109	1101101	155	6d	1.0.1001	m
110	1101110	156	6e	1.1.0	n
111	1101111	157	6f	1.1.1	o
112	1110000	160	70	1.1.10	p
113	1110001	161	71	1.1.11	q
114	1110010	162	72	1.1.100	r
115	1110011	163	73	1.1.101	s
116	1110100	164	74	1.1.110	t
117	1110101	165	75	1.1.111	u
118	1110110	166	76	1.1.1000	v
119	1110111	167	77	1.1.1001	w
120	1111000	170	78	1.10.0	x
121	1111001	171	79	1.10.1	y
122	1111010	172	7a	1.10.10	z
123	1111011	173	7b	1.10.11	{
124	1111100	174	7c	1.10.100	|
125	1111101	175	7d	1.10.101	}
126	1111110	176	7e	1.10.110	~
127	1111111	177	7f	1.10.111	del
128	10000000	200	80	1.10.1000	
129	10000001	201	81	1.10.1001	
130	10000010	202	82	1.11.0	
131	10000011	203	83	1.11.1	
132	10000100	204	84	1.11.10	
133	10000101	205	85	1.11.11	
134	10000110	206	86	1.11.100	
135	10000111	207	87	1.11.101	
136	10001000	210	88	1.11.110	
137	10001001	211	89	1.11.111	
138	10001010	212	8a	1.11.1000	
139	10001011	213	8b	1.11.1001	
140	10001100	214	8c	1.100.0	
141	10001101	215	8d	1.100.1	
142	10001110	216	8e	1.100.10	
143	10001111	217	8f	1.100.11	
144	10010000	220	90	1.100.100	
145	10010001	221	91	1.100.101	
146	10010010	222	92	1.100.110	
147	10010011	223	93	1.100.111	
148	10010100	224	94	1.100.1000	
149	10010101	225	95	1.100.1001	
150	10010110	226	96	1.101.0	
151	10010111	227	97	1.101.1	
152	10011000	230	98	1.101.10	
153	10011001	231	99	1.101.11	

154	10011010	232	9a	1.101.100
155	10011011	233	9b	1.101.101
156	10011100	234	9c	1.101.110
157	10011101	235	9d	1.101.111
158	10011110	236	9e	1.101.1000
159	10011111	237	9f	1.101.1001
160	10100000	240	a0	1.110.0
161	10100001	241	a1	1.110.1
162	10100010	242	a2	1.110.10
163	10100011	243	a3	1.110.11
164	10100100	244	a4	1.110.100
165	10100101	245	a5	1.110.101
166	10100110	246	a6	1.110.110
167	10100111	247	a7	1.110.111
168	10101000	250	a8	1.110.1000
169	10101001	251	a9	1.110.1001
170	10101010	252	aa	1.111.0
171	10101011	253	ab	1.111.1
172	10101100	254	ac	1.111.10
173	10101101	255	ad	1.111.11
174	10101110	256	ae	1.111.100
175	10101111	257	af	1.111.101
176	10110000	260	b0	1.111.110
177	10110001	261	b1	1.111.111
178	10110010	262	b2	1.111.1000
179	10110011	263	b3	1.111.1001
180	10110100	264	b4	1.1000.0
181	10110101	265	b5	1.1000.1
182	10110110	266	b6	1.1000.10
183	10110111	267	b7	1.1000.11
184	10111000	270	b8	1.1000.100
185	10111001	271	b9	1.1000.101
186	10111010	272	ba	1.1000.110
187	10111011	273	bb	1.1000.111
188	10111100	274	bc	1.1000.1000
189	10111101	275	bd	1.1000.1001
190	10111110	276	be	1.1001.0
191	10111111	277	bf	1.1001.1
192	11000000	300	c0	1.1001.10
193	11000001	301	c1	1.1001.11
194	11000010	302	c2	1.1001.100
195	11000011	303	c3	1.1001.101
196	11000100	304	c4	1.1001.110
197	11000101	305	c5	1.1001.111
198	11000110	306	c6	1.1001.1000
199	11000111	307	c7	1.1001.1001
200	11001000	310	c8	10.0.0
201	11001001	311	c9	10.0.1
202	11001010	312	ca	10.0.10
203	11001011	313	cb	10.0.11
204	11001100	314	cc	10.0.100
205	11001101	315	cd	10.0.101
206	11001110	316	ce	10.0.110
207	11001111	317	cf	10.0.111
208	11010000	320	d0	10.0.1000
209	11010001	321	d1	10.0.1001
210	11010010	322	d2	10.1.0

211	11010011	323	d3	10.1.1
212	11010100	324	d4	10.1.10
213	11010101	325	d5	10.1.11
214	11010110	326	d6	10.1.100
215	11010111	327	d7	10.1.101
216	11011000	330	d8	10.1.110
217	11011001	331	d9	10.1.111
218	11011010	332	da	10.1.1000
219	11011011	333	db	10.1.1001
220	11011100	334	dc	10.10.0
221	11011101	335	dd	10.10.1
222	11011110	336	de	10.10.10
223	11011111	337	df	10.10.11
224	11100000	340	e0	10.10.100
225	11100001	341	e1	10.10.101
226	11100010	342	e2	10.10.110
227	11100011	343	e3	10.10.111
228	11100100	344	e4	10.10.1000
229	11100101	345	e5	10.10.1001
230	11100110	346	e6	10.11.0
231	11100111	347	e7	10.11.1
232	11101000	350	e8	10.11.10
233	11101001	351	e9	10.11.11
234	11101010	352	ea	10.11.100
235	11101011	353	eb	10.11.101
236	11101100	354	ec	10.11.110
237	11101101	355	ed	10.11.111
238	11101110	356	ee	10.11.1000
239	11101111	357	ef	10.11.1001
240	11110000	360	f0	10.100.0
241	11110001	361	f1	10.100.1
242	11110010	362	f2	10.100.10
243	11110011	363	f3	10.100.11
244	11110100	364	f4	10.100.100
245	11110101	365	f5	10.100.101
246	11110110	366	f6	10.100.110
247	11110111	367	f7	10.100.111
248	11111000	370	f8	10.100.1000
249	11111001	371	f9	10.100.1001
250	11111010	372	fa	10.101.0
251	11111011	373	fb	10.101.1
252	11111100	374	fc	10.101.10
253	11111101	375	fd	10.101.11
254	11111110	376	fe	10.101.100
255	11111111	377	ff	10.101.101

B Zeichensätze

B.1 EBCDIC, ASCII, Roman8, IBM-PC

Die Zeichensätze sind in den Ein- und Ausgabegeräten (Terminal, Drucker) gespeicherte Tabellen, die die Zeichen in Zahlen und zurück umsetzen.

dezimal	oktal	EBCDIC	ASCII-7	Roman8	IBM-PC
0	0	nul	nul	nul	nul
1	1	soh	soh	soh	Grafik
2	2	stx	stx	stx	Grafik
3	3	etx	etx	etx	Grafik
4	4	pf	eot	eot	Grafik
5	5	ht	enq	enq	Grafik
6	6	lc	ack	ack	Grafik
7	7	del	bel	bel	bel
8	10		bs	bs	Grafik
9	11	rlf	ht	ht	ht
10	12	smm	lf	lf	lf
11	13	vt	vt	vt	home
12	14	ff	ff	ff	ff
13	15	cr	cr	cr	cr
14	16	so	so	so	Grafik
15	17	si	si	si	Grafik
16	20	dle	dle	dle	Grafik
17	21	dc1	dc1	dc1	Grafik
18	22	dc2	dc2	dc2	Grafik
19	23	dc3	dc3	dc3	Grafik
20	24	res	dc4	dc4	Grafik
21	25	nl	nak	nak	Grafik
22	26	bs	syn	syn	Grafik
23	27	il	etb	etb	Grafik
24	30	can	can	can	Grafik
25	31	em	em	em	Grafik
26	32	cc	sub	sub	Grafik
27	33		esc	esc	Grafik
28	34	ifs	fs	fs	cur right
29	35	igs	gs	gs	cur left
30	36	irs	rs	rs	cur up
31	37	ius	us	us	cur down
32	40	ds	space	space	space
33	41	sos	!	!	!
34	42	fs	”	”	”
35	43		#	#	#
36	44	byp	$	$	$
37	45	lf	%	%	%
38	46	etb	&	&	&
39	47	esc	’	’	’
40	50		(	(	(
41	51		)	)	)
42	52	sm	*	*	*
43	53		+	+	+

44	54		,	,	,
45	55	enq	-	-	-
46	56	ack	.	.	.
47	57	bel	/	/	/
48	60		0	0	0
49	61		1	1	1
50	62	syn	2	2	2
51	63		3	3	3
52	64	pn	4	4	4
53	65	rs	5	5	5
54	66	uc	6	6	6
55	67	eot	7	7	7
56	70		8	8	8
57	71		9	9	9
58	72		:	:	:
59	73		;	;	;
60	74	dc4	<	<	<
61	75	nak	=	=	=
62	76		>	>	>
63	77	sub	?	?	?
64	100	space	@	@	@
65	101		A	A	A
66	102	â	B	B	B
67	103	ä	C	C	C
68	104	à	D	D	D
69	105	á	E	E	E
70	106	ã	F	F	F
71	107	å	G	G	G
72	110	ç	H	H	H
73	111	ñ	I	I	I
74	112	[	J	J	J
75	113	.	K	K	K
76	114	<	L	L	L
77	115	(	M	M	M
78	116	+	N	N	N
79	117	!	O	O	O
80	120	&	P	P	P
81	121	é	Q	Q	Q
82	122	ê	R	R	R
83	123	ë	S	S	S
84	124	è	T	T	T
85	125	í	U	U	U
86	126	î	V	V	V
87	127	ï	W	W	W
88	130	ì	X	X	X
89	131	ß	Y	Y	Y
90	132	]	Z	Z	Z
91	133	$	[	[	[
92	134	*	\	\	\
93	135	)	]	]	]
94	136	;	^	^	^
95	137	^	_	_	_
96	140	–	‘	‘	‘
97	141	/	a	a	a
98	142	Â	b	b	b
99	143	Ä	c	c	c

100	144	À	d	d	d
101	145	Á	e	e	e
102	146	Ã	f	f	f
103	147	Å	g	g	g
104	150	Ç	h	h	h
105	151	Ñ	i	i	i
106	152	|	j	j	j
107	153	,	k	k	k
108	154	%	l	l	l
109	155	_	m	m	m
110	156	>	n	n	n
111	157	?	o	o	o
112	160	ø	p	p	p
113	161	É	q	q	q
114	162	Ê	r	r	r
115	163	Ë	s	s	s
116	164	È	t	t	t
117	165	Í	u	u	u
118	166	Î	v	v	v
119	167	Ï	w	w	w
120	170	Ì	x	x	x
121	171	‘	y	y	y
122	172	:	z	z	z
123	173	#	{	{	{
124	174	@	|	|	|
125	175	’	}	}	}
126	176	=	~	~	~
127	177	”	del	del	Grafik
128	200	Ø			Ç
129	201	a			ü
130	202	b			é
131	203	c			â
132	204	d			ä
133	205	e			à
134	206	f			å
135	207	g			ç
136	210	h			ê
137	211	i			ë
138	212	≪			è
139	213	≫			ı
140	214				î
141	215	ý			ì
142	216				Ä
143	217	±			Å
144	220				É
145	221	j			œ
146	222	k			Æ
147	223	l			ô
148	224	m			ö
149	225	n			ò
150	226	o			û
151	227	p			ù
152	230	q			y
153	231	r			Ö
154	232	a̲			Ü

155	233	º		
156	234	æ		£
157	235	–		Yen
158	236	Æ		Pt
159	237			*f*
160	240	μ		á
161	241	$\sim$	À	í
162	242	s	Â	ó
163	243	t	È	ú
164	244	u	Ê	ñ
165	245	v	Ë	Ñ
166	246	w	Î	ª
167	247	x	Ï	º
168	250	y	’	¿
169	251	z	‘	Grafik
170	252	¡	^	Grafik
171	253	¿		1/2
172	254		~	1/4
173	255	Ý	Ù	¡
174	256		Û	$\ll$
175	257			$\gg$
176	260			Grafik
177	261	£		Grafik
178	262	Yen		Grafik
179	263		○	Grafik
180	264	*f*	Ç	Grafik
181	265	§	ç	Grafik
182	266	¶	Ñ	Grafik
183	267		ñ	Grafik
184	270		¡	Grafik
185	271		¿	Grafik
186	272			Grafik
187	273	|	£	Grafik
188	274	–	Yen	Grafik
189	275		§	Grafik
190	276		*f*	Grafik
191	277	=		Grafik
192	300	{	â	Grafik
193	301	A	ê	Grafik
194	302	B	ô	Grafik
195	303	C	û	Grafik
196	304	D	á	Grafik
197	305	E	é	Grafik
198	306	F	ó	Grafik
199	307	G	ú	Grafik
200	310	H	à	Grafik
201	311	I	è	Grafik
202	312		ò	Grafik
203	313	ô	ù	Grafik
204	314	ö	ä	Grafik
205	315	ò	ë	Grafik
206	316	ó	ö	Grafik
207	317	õ	ü	Grafik
208	320	}	Å	Grafik
209	321	J	î	Grafik
210	322	K	Ø	Grafik

211	323	L	Æ	Grafik
212	324	M	å	Grafik
213	325	N	í	Grafik
214	326	O	ø	Grafik
215	327	P	æ	Grafik
216	330	Q	Ä	Grafik
217	331	R	ì	Grafik
218	332		Ö	Grafik
219	333	û	Ü	Grafik
220	334	ü	É	Grafik
221	335	ù	ı	Grafik
222	336	ú	ß	Grafik
223	337	y	Ô	Grafik
224	340	\	Á	α
225	341		Ã	β
226	342	S	ã	Γ
227	343	T		π
228	344	U		Σ
229	345	V	Í	σ
230	346	W	Ì	μ
231	347	X	Ó	τ
232	350	Y	Ò	Φ
233	351	Z	Õ	θ
234	352		õ	Ω
235	353	Ô	Š	δ
236	354	Ö	š	∞
237	355	Ò	Ú	Ø
238	356	Ó	Y	$\in$
239	357	Õ	y	$\cap$
240	360	0	thorn	$\equiv$
241	361	1	Thorn	$\pm$
242	362	2		$\geq$
243	363	3		$\leq$
244	364	4		Haken
245	365	5		Haken
246	366	6	–	$\div$
247	367	7	1/4	$\approx$
248	370	8	1/2	$\circ$
249	371	9	a̲	$\bullet$
250	372		o̲	$\cdot$
251	373	Û	$\ll$	$\surd$
252	374	Ü	$\sqcup$	n
253	375	Ù	$\gg$	2
254	376	Ú	$\pm$	$\sqcup$
255	377			(FF)

B.2 German-ASCII

Falls das Ein- oder Ausgabegerät einen deutschen 7-Bit-ASCII-Zeichensatz enthält, sind folgende Ersetzungen der amerikanischen Zeichen durch deutsche Sonderzeichen üblich:

Nr.	US-Zeichen	US-ASCII	German ASCII
91	linke eckige Klammer	[	Ä
92	Backslash	\	Ö

93	rechte eckige Klammer	]	Ü
123	linke geschweifte Klammer	{	ä
124	senkrechter Strich	|	ö
125	rechte geschweifte Klammer	}	ü
126	Tilde	~	ß

Achtung: Der IBM-PC und Ausgabegeräte von Hewlett-Packard verwenden keinen 7-Bit-ASCII-Zeichensatz, sondern eigene 8-Bit-Zeichensätze, die die Sonderzeichen unter Nummern höher 127 enthalten, siehe vorhergehende Tabelle.

B.3 ASCII-Steuerzeichen

Die Steuerzeichen der Zeichensätze dienen der Übermittlung von Befehlen und Informationen an das empfangende Gerät und nicht der Ausgabe eines sicht- oder druckbaren Zeichens. Die Ausgabegeräte kennen in der Regel jedoch einen Modus (transparent, Monitor, Display Functions), in der die Steuerzeichen nicht ausgeführt, sondern angezeigt werden. Die meisten Steuerzeichen belegen keine eigene Taste auf der Tastatur, sondern werden als Kombination aus der control-Taste und einer Zeichentaste eingegeben. In C/C++ läßt sich jedes Zeichen durch seine oktale Nummer in der Form `\123` oder durch seine hexadezimale Nummer in der Form `\x53` eingeben (hier das `S`).

dezimal	C-Konst.	ASCII	Bedeutung	Tasten
0	`\x00`	nul	ASCII-Null	control @
1		soh	Start of heading	control a
2		stx	Start of text	control b
3		etx	End of text	control c
4		eot	End of transmission	control d
5		enq	Enquiry	control e
6		ack	Acknowledge	control f
7	`\a`	bel	Bell	control g
8	`\b`	bs	Backspace	control h, BS
9	`\t`	ht	Horizontal tab	control i, TAB
10	`\n`	lf	Line feed	control j, LF
11	`\v`	vt	Vertical tab	control k
12	`\f`	ff	Form feed	control l
13	`\r`	cr	Carriage return	control m, RETURN
14		so	Shift out	control n
15		si	Shift in	control o
16		dle	Data link escape	control p
17		dc1	Device control 1, xon	control q
18		dc2	Device control 2, tape	control r
19		dc3	Device control 3, xoff	control s
20		dc4	Device control 4, tape	control t
21		nak	Negative acknowledge	control u
22		syn	Synchronous idle	control v
23		etb	End transmission block	control w
24		can	Cancel	control x
25		em	End of medium	control y
26		sub	Substitute	control z
27	`\x1b`	esc	Escape	control [, ESC
28		fs	File separator	control \
29		gs	Group separator	control]
30		rs	Record separator	control ^
31		us	Unit separator	control _
127		del	Delete	DEL, RUBOUT

B.4 Latin-1 (ISO 8859-1)

Die internationale Norm ISO 8859 beschreibt gegenwärtig zehn Zeichensätze, die jedes Zeichen durch jeweils ein Byte darstellen. Jeder Zeichensatz umfaßt also maximal 256 druckbare Zeichen und Steuerzeichen. Der erste – Latin-1 genannt – ist für west- und mitteleuropäische Sprachen – darunter Deutsch – vorgesehen. Latin-2 deckt Mittel- und Osteuropa ab, soweit das lateinische Alphabet verwendet wird. Wer einen polnisch-deutschen Text schreiben will, braucht Latin 2. Die deutschen Sonderzeichen liegen in Latin 1 bis 6 an denselben Stellen. Weiteres siehe in der ISO-Norm und im RFC 1345 *Character Mnemonics and Character Sets* vom Juni 1992. Auch `http://wwwwbs.cs.tu-berlin.de/~czyborra/charsets/` hilft weiter.

Die erste Hälfte (0 – 127) aller Latin-Zeichensätze stimmt mit US-ASCII überein, die zweite mit keinem der anderen Zeichensätze. Zu jedem Zeichen gehört eine standardisierte verbale Bezeichnung. Einige Zeichen wie das isländische Thorn oder das Cent-Zeichen konnten hier mit LaTeX nicht dargestellt werden.

dezimal	oktal	hex	Zeichen	Bezeichnung
000	000	00	nu	Null (nul)
001	001	01	sh	Start of heading (soh)
002	002	02	sx	Start of text (stx)
003	003	03	ex	End of text (etx)
004	004	04	et	End of transmission (eot)
005	005	05	eq	Enquiry (enq)
006	006	06	ak	Acknowledge (ack)
007	007	07	bl	Bell (bel)
008	010	08	bs	Backspace (bs)
009	011	09	ht	Character tabulation (ht)
010	012	0a	lf	Line feed (lf)
011	013	0b	vt	Line tabulation (vt)
012	014	0c	ff	Form feed (ff)
013	015	0d	cr	Carriage return (cr)
014	016	0e	so	Shift out (so)
015	017	0f	si	Shift in (si)
016	020	10	dl	Datalink escape (dle)
017	021	11	d1	Device control one (dc1)
018	022	12	d2	Device control two (dc2)
019	023	13	d3	Device control three (dc3)
020	024	14	d4	Device control four (dc4)
021	025	15	nk	Negative acknowledge (nak)
022	026	16	sy	Synchronous idle (syn)
023	027	17	eb	End of transmission block (etb)
024	030	18	cn	Cancel (can)
025	031	19	em	End of medium (em)
026	032	1a	sb	Substitute (sub)
027	033	1b	ec	Escape (esc)
028	034	1c	fs	File separator (is4)
029	035	1d	gs	Group separator (is3)
030	036	1e	rs	Record separator (is2)
031	037	1f	us	Unit separator (is1)
032	040	20	sp	Space
033	041	21	!	Exclamation mark
034	042	22	”	Quotation mark
035	043	23	#	Number sign
036	044	24	$	Dollar sign
037	045	25	%	Percent sign
038	046	26	&	Ampersand

039	047	27	’	Apostrophe
040	050	28	(	Left parenthesis
041	051	29	)	Right parenthesis
042	052	2a	*	Asterisk
043	053	2b	+	Plus sign
044	054	2c	,	Comma
045	055	2d	-	Hyphen-Minus
046	056	2e	.	Full stop
047	057	2f	/	Solidus
048	060	30	0	Digit zero
049	061	31	1	Digit one
050	062	32	2	Digit two
051	063	33	3	Digit three
052	064	34	4	Digit four
053	065	35	5	Digit five
054	066	36	6	Digit six
055	067	37	7	Digit seven
056	070	38	8	Digit eight
057	071	39	9	Digit nine
058	072	3a	:	Colon
059	073	3b	;	Semicolon
060	074	3c	<	Less-than sign
061	075	3d	=	Equals sign
062	076	3e	>	Greater-than sign
063	077	3f	?	Question mark
064	100	40	@	Commercial at
065	101	41	A	Latin capital letter a
066	102	42	B	Latin capital letter b
067	103	43	C	Latin capital letter c
068	104	44	D	Latin capital letter d
069	105	45	E	Latin capital letter e
070	106	46	F	Latin capital letter f
071	107	47	G	Latin capital letter g
072	110	48	H	Latin capital letter h
073	111	49	I	Latin capital letter i
074	112	4a	J	Latin capital letter j
075	113	4b	K	Latin capital letter k
076	114	4c	L	Latin capital letter l
077	115	4d	M	Latin capital letter m
078	116	4e	N	Latin capital letter n
079	117	4f	O	Latin capital letter o
080	120	50	P	Latin capital letter p
081	121	51	Q	Latin capital letter q
082	122	52	R	Latin capital letter r
083	123	53	S	Latin capital letter s
084	124	54	T	Latin capital letter t
085	125	55	U	Latin capital letter u
086	126	56	V	Latin capital letter v
087	127	57	W	Latin capital letter w
088	130	58	X	Latin capital letter x
089	131	59	Y	Latin capital letter y
090	132	5a	Z	Latin capital letter z
091	133	5b	[	Left square bracket
092	134	5c	\	Reverse solidus
093	135	5d	]	Right square bracket
094	136	5e	^	Circumflex accent
095	137	5f	_	Low line

096	140	60	‘	Grave accent
097	141	61	a	Latin small letter a
098	142	62	b	Latin small letter b
099	143	63	c	Latin small letter c
100	144	64	d	Latin small letter d
101	145	65	e	Latin small letter e
102	146	66	f	Latin small letter f
103	147	67	g	Latin small letter g
104	150	68	h	Latin small letter h
105	151	69	i	Latin small letter i
106	152	6a	j	Latin small letter j
107	153	6b	k	Latin small letter k
108	154	6c	l	Latin small letter l
109	155	6d	m	Latin small letter m
110	156	6e	n	Latin small letter n
111	157	6f	o	Latin small letter o
112	160	70	p	Latin small letter p
113	161	71	q	Latin small letter q
114	162	72	r	Latin small letter r
115	163	73	s	Latin small letter s
116	164	74	t	Latin small letter t
117	165	75	u	Latin small letter u
118	166	76	v	Latin small letter v
119	167	77	w	Latin small letter w
120	170	78	x	Latin small letter x
121	171	79	y	Latin small letter y
122	172	7a	z	Latin small letter z
123	173	7b	{	Left curly bracket
124	174	7c	|	Vertical line
125	175	7d	}	Right curly bracket
126	176	7e	~	Tilde
127	177	7f	dt	Delete (del)
128	200	80	pa	Padding character (pad)
129	201	81	ho	High octet preset (hop)
130	202	82	bh	Break permitted here (bph)
131	203	83	nh	No break here (nbh)
132	204	84	in	Index (ind)
133	205	85	nl	Next line (nel)
134	206	86	sa	Start of selected area (ssa)
135	207	87	es	End of selected area (esa)
136	210	88	hs	Character tabulation set (hts)
137	211	89	hj	Character tabulation with justification (htj)
138	212	8a	vs	Line tabulation set (vts)
139	213	8b	pd	Partial line forward (pld)
140	214	8c	pu	Partial line backward (plu)
141	215	8d	ri	Reverse line feed (ri)
142	216	8e	s2	Single-shift two (ss2)
143	217	8f	s3	Single-shift three (ss3)
144	220	90	dc	Device control string (dcs)
145	221	91	p1	Private use one (pu1)
146	222	92	p2	Private use two (pu2)
147	223	93	ts	Set transmit state (sts)
148	224	94	cc	Cancel character (cch)
149	225	95	mw	Message waiting (mw)
150	226	96	sg	Start of guarded area (spa)
151	227	97	eg	End of guarded area (epa)
152	230	98	ss	Start of string (sos)

153	231	99	gc	Single graphic character introducer (sgci)
154	232	9a	sc	Single character introducer (sci)
155	233	9b	ci	Control sequence introducer (csi)
156	234	9c	st	String terminator (st)
157	235	9d	oc	Operating system command (osc)
158	236	9e	pm	Privacy message (pm)
159	237	9f	ac	Application program command (apc)
160	240	a0	ns	No-break space
161	241	a1	¡	Inverted exclamation mark
162	242	a2		Cent sign
163	243	a3	£	Pound sign
164	244	a4		Currency sign (künftig Euro?)
165	245	a5		Yen sign
166	246	a6		Broken bar
167	247	a7	§	Section sign
168	250	a8		Diaresis
169	251	a9	©	Copyright sign
170	252	aa	^{a}	Feminine ordinal indicator
171	253	ab	$\ll$	Left-pointing double angle quotation mark
172	254	ac	$\neg$	Not sign
173	255	ad	-	Soft hyphen
174	256	ae		Registered sign
175	257	af	$^-$	Overline
176	260	b0	$^\circ$	Degree sign
177	261	b1	$\pm$	Plus-minus sign
178	262	b2	^{2}	Superscript two
179	263	b3	^{3}	Superscript three
180	264	b4	’	Acute accent
181	265	b5	μ	Micro sign
182	266	b6	¶	Pilcrow sign
183	267	b7	·	Middle dot
184	270	b8	¸	Cedilla
185	271	b9	^{1}	Superscript one
186	272	ba	$^\circ$	Masculine ordinal indicator
187	273	bb	$\gg$	Right-pointing double angle quotation mark
188	274	bc	1/4	Vulgar fraction one quarter
189	275	bd	1/2	Vulgar fraction one half
190	276	be	3/4	Vulgar fraction three quarters
191	277	bf	¿	Inverted question mark
192	300	c0	À	Latin capital letter a with grave
193	301	c1	Á	Latin capital letter a with acute
194	302	c2	Â	Latin capital letter a with circumflex
195	303	c3	Ã	Latin capital letter a with tilde
196	304	c4	Ä	Latin capital letter a with diaresis
197	305	c5	Å	Latin capital letter a with ring above
198	306	c6	Æ	Latin capital letter ae
199	307	c7	Ç	Latin capital letter c with cedilla
200	310	c8	È	Latin capital letter e with grave
201	311	c9	É	Latin capital letter e with acute
202	312	ca	Ê	Latin capital letter e with circumflex
203	313	cb	Ë	Latin capital letter e with diaresis
204	314	cc	Ì	Latin capital letter i with grave
205	315	cd	Í	Latin capital letter i with acute
206	316	ce	Î	Latin capital letter i with circumflex
207	317	cf	Ï	Latin capital letter i with diaresis

208	320	d0		Latin capital letter eth (Icelandic)
209	321	d1	Ñ	Latin capital letter n with tilde
210	322	d2	Ò	Latin capital letter o with grave
211	323	d3	Ó	Latin capital letter o with acute
212	324	d4	Ô	Latin capital letter o with circumflex
213	325	d5	Õ	Latin capital letter o with tilde
214	326	d6	Ö	Latin capital letter o with diaresis
215	327	d7	×	Multiplication sign
216	330	d8	Ø	Latin capital letter o with stroke
217	331	d9	Ù	Latin capital letter u with grave
218	332	da	Ú	Latin capital letter u with acute
219	333	db	Û	Latin capital letter u with circumflex
220	334	dc	Ü	Latin capital letter u with diaresis
221	335	dd	Ý	Latin capital letter y with acute
222	336	de		Latin capital letter thorn (Icelandic)
223	337	df	ß	Latin small letter sharp s (German)
224	340	e0	à	Latin small letter a with grave
225	341	e1	á	Latin small letter a with acute
226	342	e2	â	Latin small letter a with circumflex
227	343	e3	ã	Latin small letter a with tilde
228	344	e4	ä	Latin small letter a with diaresis
229	345	e5	å	Latin small letter a with ring above
230	346	e6	æ	Latin small letter ae
231	347	e7	ç	Latin small letter c with cedilla
232	350	e8	è	Latin small letter e with grave
233	351	e9	é	Latin small letter e with acute
234	352	ea	ê	Latin small letter e with circumflex
235	353	eb	ë	Latin small letter e with diaresis
236	354	ec	ì	Latin small letter i with grave
237	355	ed	í	Latin small letter i with acute
238	356	ee	î	Latin small letter i with circumflex
239	357	ef	ï	Latin small letter i with diaresis
240	360	f0		Latin small letter eth (Icelandic)
241	361	f1	ñ	Latin small letter n with tilde
242	362	f2	ò	Latin small letter o with grave
243	363	f3	ó	Latin small letter o with acute
244	364	f4	ô	Latin small letter o with circumflex
245	365	f5	õ	Latin small letter o with tilde
246	366	f6	ö	Latin small letter o with diaresis
247	367	f7	÷	Division sign
248	370	f8	ø	Latin small letter o with stroke
249	371	f9	ù	Latin small letter u with grave
250	372	fa	ú	Latin small letter u with acute
251	373	fb	û	Latin small letter u with circumflex
252	374	fc	ü	Latin small letter u with diaresis
253	375	fd	ý	Latin small letter y with acute
254	376	fe		Latin small letter thorn (Icelandic)
255	377	ff	ÿ	Latin small letter y with diaresis

C UNIX-Systemaufrufe

Systemaufrufe werden vom Anwendungsprogramm wie eigene oder fremde Funktionen angesehen. Ihrem Ursprung nach sind es auch C-Funktionen. Sie sind jedoch nicht Bestandteil einer Funktionsbibliothek, sondern gehören zum Betriebssystem und sind nicht durch andere Funktionen erweiterbar.

Die Systemaufrufe – als Bestandteil des Betriebssystems – sind für alle Programmiersprachen dieselben, während die Funktionsbibliotheken zur jeweiligen Programmiersprache gehören. Folgende Systemaufrufe sind unter UNIX verfügbar:

`access`	prüft Zugriff auf File
`acct`	startet und stoppt Prozess Accounting
`alarm`	setzt Weckeruhr für Prozess
`atexit`	Funktion für Programmende
`brk`	ändert Speicherzuweisung
`chdir`	wechselt Arbeitsverzeichnis
`chmod`	ändert Zugriffsrechte eines Files
`chown`	ändert Besitzer eines Files
`chroot`	ändert Root-Verzeichnis
`close`	schließt einen File-Deskriptor
`creat`	öffnet File, ordnet Deskriptor zu
`dup`	dupliziert File-Deskriptor
`errno`	Fehlervariable der Systemaufrufe
`exec`	führt ein Programm aus
`exit`	beendet einen Prozess
`fcntl`	Filesteuerung
`fork`	erzeugt einen neuen Prozess
`fsctl`	liest Information aus File-System
`fsync`	schreibt File aus Arbeitsspeicher auf Platte
`getaccess`	ermittelt Zugriffsrechte
`getacl`	ermittelt Zugriffsrechte
`getcontext`	ermittelt Kontext eines Prozesses
`getdirentries`	ermittelt Verzeichnis-Einträge
`getgroups`	ermittelt Gruppenrechte eines Prozesses
`gethostname`	ermittelt Namen des Systems
`getitimer`	setzt oder liest Intervall-Uhr
`getpid`	liest Prozess-ID
`gettimeofday`	ermittelt Zeit
`getuid`	liest User-ID des aufrufenden Prozesses
`ioctl`	I/O-Steuerung
`kill`	schickt Signal an einen Prozess
`link`	linkt ein File
`lockf`	setzt Semaphore und Record-Sperren
`lseek`	bewegt Schreiblesezeiger in einem File
`mkdir`	erzeugt Verzeichnis
`mknod`	erzeugt File
`mount`	hängt File-System in File-Hierarchie ein
`msgctl`	Interprozess-Kommunikation
`nice`	ändert die Priorität eines Prozesses
`open`	öffnet File zum Lesen oder Schreiben
`pause`	suspendiert Prozess bis zum Empfang eines Signals
`pipe`	erzeugt eine Pipe
`prealloc`	reserviert Arbeitsspeicher

`profil`	ermittelt Zeiten bei der Ausführung eines Programmes
`read`	liest aus einem File
`readlink`	liest symbolisches Link
`rename`	ändert Filenamen
`rmdir`	löscht Verzeichnis
`rtprio`	ändert Echtzeit-Priorität
`semctl`	Semaphore
`setgrp`	setzt Gruppen-Zugriffsrechte eines Prozesses
`setuid`	setzt User-ID eines Prozesses
`signal`	legt fest, was auf ein Signal hin zu tun ist
`stat`	liest die Inode eines Files
`statfs`	liest Werte des File-Systems
`symlink`	erzeugt symbolischen Link
`sync`	schreibt Puffer auf Platte
`szsconf`	ermittelt Systemwerte
`time`	ermittelt die Systemzeit
`times`	ermittelt Zeitverbrauch eines Prozesses
`truncate`	schneidet File ab
`umask`	setzt oder ermittelt Filezugriffsmaske
`umount`	entfernt Filesystem aus File-Hierarchie
`unlink`	löscht File
`ustat`	liest Werte des File-Systems
`utime`	setzt Zeitstempel eines Files
`wait`	wartet auf Ende eines Kindprozesses
`write`	schreibt in ein File

Die Aufzählung kann durch weitere Systemaufrufe des jeweiligen Lieferanten des Betriebssystems (z. B. Hewlett-Packard) ergänzt werden. Diese erleichtern das Programmieren, verschlechtern aber die Portabilität. Zu den meisten Systemaufrufen mit `get...` gibt es ein Gegenstück `set...`, das in einigen Fällen dem Superuser vorbehalten ist.

D C-Lexikon

D.1 Schlüsselwörter

In C/C++ dürfen Schlüsselwörter keinesfalls als Namen verwendet werden. Laut ANSI verwendet C folgende Schlüsselwörter (Wortsymbole, keywords):

- Deklaratoren
  - `auto`, Default-Speicherklasse (kann weggelassen werden)
  - `char`, Zeichentyp
  - `const`, Typattribut (neu in ANSI-C)
  - `double`, Typ Gleitkommazahl doppelter Genauigkeit
  - `enum`, Aufzählungstyp
  - `extern`, Speicherklasse
  - `float`, Typ Gleitkommazahl einfacher Genauigkeit
  - `int`, Typ Ganzzahl einfacher Länge
  - `long`, Typ Ganzzahl doppelter Länge
  - `register`, Speicherklasse Registervariable
  - `short`, Typ Ganzzahl halber Länge
  - `signed`, Typzusatz zu Ganzzahl oder Zeichen
  - `static`, Speicherklasse
  - `struct`, Strukturtyp
  - `typedef`, Definition eines benutzereigenen Typs
  - `union`, Typ Union
  - `unsigned`, Typzusatz zu Ganzzahl oder Zeichen
  - `void`, leerer Typ
  - `volatile`, Typattribut (neu in ANSI-C)
- Schleifen und Bedingungen (Kontrollanweisungen)
  - `break`, Verlassen einer Schleife
  - `case`, Fall einer Auswahl (`switch`)
  - `continue`, Rücksprung vor eine Schleife
  - `default`, Default-Fall einer Auswahl (`switch`)
  - `do`, Beginn einer `do`-Schleife
  - `else`, Alternative einer Verzweigung
  - `for`, Beginn einer `for`-Schleife
  - `goto`, unbedingter Sprung
  - `if`, Bedingung oder Beginn einer Verzweigung
  - `switch`, Beginn einer Auswahl
  - `while`, Beginn einer `while`-Schleife
- Sonstige
  - `return`, Rücksprung in die aufrufende Einheit

  - `sizeof`, Bytebedarf eines Typs oder einer Variablen

In C++ kommen laut BJARNE STROUSTRUP hinzu:

- `catch`, Ausnahmebehandlung
- `class`, Klassendeklaration
- `delete`, Löschen eines Objektes
- `friend`, Deklaration einer Funktion
- `inline`, inline-Funktion
- `new`, Erzeugen eines Objektes
- `operator`, Überladen von Operatoren
- `private`, Deklaration von Klassenmitgliedern
- `protected`, Deklaration von Klassenmitgliedern
- `public`, Deklaration von Klassenmitgliedern
- `template`, Deklaration eines Templates (Klasse)
- `this`, Pointer auf Objekt
- `throw`, Ausnahmebehandlung
- `try`, Ausnahmebehandlung
- `virtual`, Deklaration

Darüber hinaus verwenden einige Compiler weitere Schlüsselwörter:

- `asm`, Assembler-Aufruf innerhalb einer C- oder C++-Quelle
- `bool`, logischer oder boolescher Typ
- `cdecl`, Aufruf einer Funktion nach C-Konventionen
- `const_cast`, cast-Operator für const-Werte
- `dynamic_cast`, cast-Operator
- `entry`, (war in K&R-C für künftigen Gebrauch vorgesehen)
- `explicit`, Konstruktor-Vereinbarung
- `export`, Vereinbarung bei Klassen-Templates
- `false`, boolesche Konstante
- `far`, Typzusatz unter MS-DOS
- `fortran`, Aufruf einer Funktion nach FORTRAN-Konventionen
- `huge`, Typzusatz unter MS-DOS
- `mutable`, Typattribut
- `namespace`, Vereinbarung des Geltungsbereiches von Namen
- `near`, Typzusatz unter MS-DOS
- `pascal`, Aufruf einer Funktion nach PASCAL-Konventionen
- `reinterpret_cast`, cast-Operator
- `static_cast`, cast-Operator
- `true`, boolesche Konstante
- `typeid`, Operator zum Ermitteln des Typs
- `typename`, Alternative zum Schlüsselwort `class`
- `using`, Deklaration in Verbindung mit `namespace`
- `wchar_t`, Typ (wide character literal)

D.2 Operatoren

Die Operatoren von C/C++ sind im folgenden ihrem Vorrang nach geordnet, höchster Rang (Bindungskraft) zuoberst. Alle Operatoren eines Abschnitts haben gleichen Rang. l bzw. r bedeutet von links bzw. rechts her assoziativ. Ein unärer Operator verlangt einen Operanden, ein binärer zwei und ein ternärer drei.

<table>
<tr><th>Operator</th><th>A</th><th>-</th><th>Bedeutung</th></tr>
<tr><td>::</td><td>r</td><td>unär</td><td>Bezugsrahmen, global</td></tr>
<tr><td>::</td><td>l</td><td>binär</td><td>Bezugsrahmen, Klasse</td></tr>
<tr><td>::</td><td>l</td><td>binär</td><td>Bezugsrahmen, Namensraum</td></tr>
<tr><td>( )</td><td>l</td><td>unär</td><td>Klammerung, Funktion</td></tr>
<tr><td>[ ]</td><td>l</td><td>unär</td><td>Index</td></tr>
<tr><td>-></td><td>l</td><td>binär</td><td>Auswahl</td></tr>
<tr><td>.</td><td>l</td><td>binär</td><td>Auswahl</td></tr>
<tr><td>++</td><td>r</td><td>unär</td><td>Postfix Inkrement</td></tr>
<tr><td>--</td><td>r</td><td>unär</td><td>Postfix Dekrement</td></tr>
<tr><td>typeid</td><td></td><td>unär</td><td>Typabfrage</td></tr>
<tr><td>const_cast</td><td></td><td>binär</td><td>Typumwandlung</td></tr>
<tr><td>dynamic_cast</td><td></td><td>binär</td><td>Typumwandlung</td></tr>
<tr><td>reinterpret_cast</td><td></td><td>binär</td><td>Typumwandlung</td></tr>
<tr><td>static_cast</td><td></td><td>binär</td><td>Typumwandlung</td></tr>
<tr><td>sizeof</td><td>r</td><td>unär</td><td>Größenabfrage</td></tr>
<tr><td>++</td><td>r</td><td>unär</td><td>Präfix Inkrement</td></tr>
<tr><td>--</td><td>r</td><td>unär</td><td>Präfix Dekrement</td></tr>
<tr><td>~</td><td>r</td><td>unär</td><td>bitweise Negation</td></tr>
<tr><td>!</td><td>r</td><td>unär</td><td>logische Negation</td></tr>
<tr><td>-</td><td>r</td><td>unär</td><td>negatives Vorzeichen</td></tr>
<tr><td>+</td><td>r</td><td>unär</td><td>positives Vorzeichen</td></tr>
<tr><td>*</td><td>r</td><td>unär</td><td>Dereferenzierung</td></tr>
<tr><td>&</td><td>r</td><td>unär</td><td>Referenzierung</td></tr>
<tr><td>( )</td><td>r</td><td>binär</td><td>cast-Operator</td></tr>
<tr><td>new</td><td>r</td><td>unär</td><td>dynamische Speicherbelegung</td></tr>
<tr><td>delete</td><td>r</td><td>unär</td><td>dynamische Speicherfreigabe</td></tr>
<tr><td>->*</td><td>l</td><td>binär</td><td>Auswahl</td></tr>
<tr><td>.*</td><td>l</td><td>binär</td><td>Auswahl</td></tr>
<tr><td>*</td><td>l</td><td>binär</td><td>Multiplikation</td></tr>
<tr><td>/</td><td>l</td><td>binär</td><td>Division</td></tr>
<tr><td>%</td><td>l</td><td>binär</td><td>Modulus (Divisionsrest)</td></tr>
<tr><td>+</td><td>l</td><td>binär</td><td>Addition</td></tr>
<tr><td>-</td><td>l</td><td>binär</td><td>Subtraktion</td></tr>
<tr><td><<</td><td>l</td><td>binär</td><td>bitweises Shiften links</td></tr>
<tr><td>>></td><td>l</td><td>binär</td><td>bitweises Shiften rechts</td></tr>
<tr><td><</td><td>l</td><td>binär</td><td>kleiner als</td></tr>
<tr><td><=</td><td>l</td><td>binär</td><td>kleiner gleich</td></tr>
<tr><td>></td><td>l</td><td>binär</td><td>größer als</td></tr>
<tr><td>>=</td><td>l</td><td>binär</td><td>größer gleich</td></tr>
<tr><td>==</td><td>l</td><td>binär</td><td>Gleichheit</td></tr>
<tr><td>!=</td><td>l</td><td>binär</td><td>Ungleichheit</td></tr>
<tr><td>&</td><td>l</td><td>binär</td><td>bitweises Und</td></tr>
<tr><td>^</td><td>l</td><td>binär</td><td>bitweises exklusives Oder</td></tr>
<tr><td>|</td><td>l</td><td>binär</td><td>bitweises Oder</td></tr>
<tr><td>&&</td><td>l</td><td>binär</td><td>logisches Und</td></tr>
<tr><td>||</td><td>l</td><td>binär</td><td>logisches Oder</td></tr>
<tr><td>?:</td><td>l</td><td>ternär</td><td>bedingte Bewertung</td></tr>
<tr><td>=</td><td>r</td><td>binär</td><td>Zuweisung</td></tr>
</table>

<table>
<tr><td>+=, -=, *=, /=
%=, >>=, <<=,
&=, |=, ^=</td><td>r</td><td>binär</td><td>zusammengesetzte Zuweisung</td></tr>
<tr><td>throw</td><td></td><td>unär</td><td>Ausnahmebehandlung</td></tr>
<tr><td>,</td><td>l</td><td></td><td>Komma-Operator</td></tr>
</table>

D.3 Standardfunktionen

Folgende Standardfunktionen oder -makros sind gebräuchlich:

- Pufferbehandlung
  - `memchr`, sucht Zeichen im Puffer
  - `memcmp`, vergleicht Zeichen mit Pufferinhalt
  - `memcpy`, kopiert Zeichen in Puffern
  - `memset`, setzt Puffer auf bestimmtes Zeichen
- Zeichenbehandlung
  - `isalnum`, prüft Zeichen, ob alphanumerisch
  - `isalpha`, prüft Zeichen, ob Buchstabe
  - `iscntrl`, prüft Zeichen, ob Kontrollzeichen
  - `isdigit`, prüft Zeichen, ob Ziffer
  - `isgraph`, prüft Zeichen, ob sichtbar
  - `islower`, prüft Zeichen, ob Kleinbuchstabe
  - `isprint`, prüft Zeichen, ob druckbar
  - `ispunct`, prüft Zeichen, ob Satzzeichen
  - `isspace`, prüft Zeichen, ob Whitespace
  - `isupper`, prüft Zeichen. ob Großbuchstabe
  - `isxdigit`, prüft Zeichen, ob hexadezimale Ziffer
  - `tolower`, wandelt Großbuchstaben in Kleinbuchstaben um
  - `toupper`, wandelt Kleinbuchstaben in Großbuchstaben um
- Datenumwandlung
  - `atof`, wandelt String in `double`-Wert um
  - `atoi`, wandelt String in `int`-Wert um
  - `atol`, wandelt String in `long`-Wert um
  - `strtod`, wandelt String in `double`-Wert um
  - `strtol`, wandelt String in `long`-Wert um
  - `strtoul`, wandelt String in `unsigned long`-Wert um
- Filebehandlung
  - `remove`, löscht File
  - `rename`, ändert Namen eines Files
- Ein- und Ausgabe
  - `clearerr`, löscht Fehlermeldung eines Filepointers
  - `fclose`, schließt Filepointer
  - `fflush`, leert Puffer eines Filepointers

  - `fgetc`, liest Zeichen von Filepointer
  - `fgetpos`, ermittelt Stand des Lesezeigers
  - `fgets`, liest String von Filepointer
  - `fopen`, öffnet Filepointer
  - `fprintf`, schreibt formatiert nach Filepointer
  - `fputc`, schreibt Zeichen nach Filepointer
  - `fputs`, schreibt String nach Filepointer
  - `fread`, liest Bytes von Filepointer
  - `freopen`, ersetzt geöffneten Filepointer
  - `fscanf`, liest formatiert von Filepointer
  - `fseek`, setzt Lesezeiger auf bestimmte Stelle
  - `fsetpos`, setzt Lesezeiger auf bestimmte Stelle
  - `ftell`, ermittelt Stellung des Lesezeigers
  - `fwrite`, schreibt Bytes nach Filepointer
  - `getc`, liest Zeichen von Filepointer
  - `getchar`, liest Zeichen von `stdin`
  - `gets`, liest String von Filepointer
  - `printf`, schreibt formatiert nach `stdout`
  - `putc`, schreibt Zeichen nach Filepointer
  - `putchar`, schreibt Zeichen nach `stdout`
  - `puts`, schreibt String nach Filepointer
  - `rewind`, setzt Lesezeiger auf Fileanfang
  - `scanf`, liest formatiert von `stdin`
  - `setbuf`, ordnet einem Filepointer einen Puffer zu
  - `setvbuf`, ordnet einem Filepointer einen Puffer zu
  - `sprintf`, schreibt formatiert in einen String
  - `sscanf`, liest formatiert aus einem String
  - `tempnam`, erzeugt einen temporären Filenamen
  - `tmpfile`, erzeugt ein temporäres File
  - `ungetc`, schreibt letztes gelesenes Zeichen zurück
  - `vfprintf`, schreibt formatiert aus einer Argumentenliste
  - `vprintf`, schreibt formatiert aus einer Argumentenliste
  - `vsprintf`, schreibt formatiert aus einer Argumentenlsite

- Mathematik
  - `acos`, arcus cosinus
  - `asin`, arcus sinus
  - `atan`, arcus tangens
  - `atan2`, arcus tangens, erweiterter Bereich
  - `ceil`, kleinste Ganzzahl
  - `cos`, cosinus
  - `cosh`, cosinus hyperbolicus
  - `exp`, Exponentialfunktion
  - `fabs`, Absolutwert, Betrag

  - `floor`, größte Ganzzahl
  - `fmod`, Divisionsrest
  - `frexp`, teilt Gleitkommazahl auf
  - `ldexp`, teilt Gleitkommazahl auf
  - `log`, Logarithmus naturalis
  - `log10`, dekadischer Logarithmus
  - `modf`, teilt Gleitkommazahl auf
  - `pow`, allgemeine Potenz
  - `sin`, sinus
  - `sinh`, sinus hyperbolicus
  - `sqrt`, positive Quadratwurzel
  - `tan`, tangens
  - `tanh`, tangens hyperbolicus

- Speicherzuweisung
  - `calloc`, allokiert Speicher für Array
  - `free`, gibt allokierten Speicher frei
  - `malloc`, allokiert Speicher
  - `realloc`, ändert Größe des allokierten Speichers

- Prozesssteuerung
  - `abort`, erzeugt SIGABRT-Signal
  - `atexit`, Funktionsaufruf bei Programmende
  - `exit`, Programmende
  - `raise`, sendet Signal
  - `signal`, legt Antwort auf Siganl fest
  - `system`, übergibt Argument an Kommandointerpreter

- Suchen und Sortieren
  - `bsearch`, binäre Suche
  - `qsort`, Quicksort

- Stringbehandlung
  - `strcat`, verkettet Strings
  - `strchr`, sucht Zeichen in String
  - `strcmp`, vergleicht Strings
  - `strcpy`, kopiert String
  - `strcspn`, sucht Teilstring
  - `strerror`, verweist auf Fehlermeldung
  - `strlen`, ermittelt Stringlänge
  - `strncat`, verkettet n Zeichen von Strings
  - `strncmp`, vergleicht n Zeichen von Strings
  - `strncpy`, kopiert n Zeichen eines Strings
  - `strpbrk`, sucht Zeichen in String
  - `strrchr`, sucht Zeichen in String
  - `strspn`, ermittelt Länge eines Teilstrings
  - `strstr`, sucht Zeichen in String

Dies sind alle Funktionen des ANSI-Vorschlags. Die meisten Compiler bieten darüberhinaus eine Vielzahl weiterer Funktionen, die das Programmieren erleichtern, aber die Portabilität verschlechtern.

D.4 printf(3), scanf(3)

`printf(3)` und `scanf(3)` sind die beiden Standardfunktionen zum Ein- und Ausgeben von Daten. Wichtiger Unterschied: `printf(3)` erwartet Variable, `scanf(3)` Pointer. Die Formatbezeichner stimmen weitgehend überein:

Bezeichner	Typ	Beispiel	Bedeutung
%c	char	a	Zeichen
%s	char *	Karlsruhe	String
%d	int	-1234	dezimale Ganzzahl mit Vorzeichen
%i	int	-1234	dezimale Ganzzahl mit Vorzeichen
%u	unsigned	1234	dezimale Ganzzahl ohne Vorzeichen
%ld	long	1234	dezimal Ganzzahl doppelter Länge
%f	double	12.34	Gleitkommazahl mit Vorzeichen
%e	double	1.234 E 1	Gleitkommazahl, Exponentialform
%g	double	12.34	kurze Darstellung von %e oder %f
%o	unsigned octal	2322	oktale Ganzzahl ohne Vorzeichen
%x	unsigned hex	4d2	hexadezimale Ganzzahl o. Vorzeichen
%p	void *	68ff32e4	Pointer
%%	-	%	Prozentzeichen

Weiteres im Referenz-Handbuch unter `printf(3)` oder `scanf(3)`. Länge, Bündigkeit, Unterdrückung führender Nullen, Vorzeichenangabe können festgelegt werden.

D.5 Include-Dateien

Die Standard-Include-Dateien enthalten in lesbarer Form Definitionen von Konstanten und Typen, Deklarationen von Funktionen und Makrodefinitionen. Sie werden von Systemaufrufen und Bibliotheksfunktionen benötigt. Bei der Beschreibung jeder Funktion im Referenz-Handbuch ist angegeben, welche Include-Dateien jeweils eingebunden werden müssen. Gebräuchliche Include-Dateien sind:

- `ctype.h`, Definition von Zeichenklassen (`conv(3)`)
- `curses.h`, Bildschirmsteuerung (`curses(3)`)
- `errno.h`, Fehlermeldungen des Systems (`errno(2)`)
- `fcntl.h`, Steuerung des Dateizugriffs (`fcntl(2), open(2)`)
- `malloc.h`, Speicherallokierung (`malloc(3)`)
- `math.h`, mathematische Funktionen (`log(3), sqrt(3), floor(3)`)
- `memory.h`, Speicherfunktionen (`memory(3)`)
- `search.h`, Suchfunktionen (`bsearch(3)`)
- `signal.h`, Signalbehandlung (`signal(2)`)
- `stdio.h`, Ein- und Ausgabe (`printf(3), scanf(3), fopen(3)`)
- `string.h`, Stringbehandlung (`string(3)`)
- `time.h`, Zeitfunktionen (`ctime(3)`)
- `varargs.h`, Argumentenliste variabler Länge (`vprintf(3)`)
- `sys/ioctl.h`, Ein- und Ausgabe (`ioctl(2)`)
- `sys/stat.h`, Zugriffsrechte (`chmod(2), mkdir(2), stat(2)`)
- `sys/types.h` , verschiedene Deklarationen (`chmod(2), getut(3)`)

Auch diese Liste ist vom Compiler und damit von der Hardware abhängig. So findet man die include-Datei `dos.h` nicht auf UNIX-Anlagen, sondern nur bei Compilern unter MS-DOS für PCs.

D.6 Präprozessor-Anweisungen

Der erste Schritt beim Compilieren ist die Bearbeitung des Quelltextes durch den Präprozessor. Dieser entfernt den Kommentar und führt Ersetzungen und Einfügungen gemäß der folgenden Anweisungen (directives) aus:

- `#define` buchstäbliche Ersetzung einer symbolischen Konstanten oder eines Makros. Ist kein Ersatz angegeben, wird nur der Name als definiert angesehen (für #ifdef). Häufig.
- `#undefine` löscht die Definition eines Namens.
- `#error` führt zu einer Fehlermeldung des Präprozessors.
- `#include` zieht die angegebene Datei herein. Häufig.
- `#if, #else, #elif, #endif` falls Bedingung zutrifft, werden die nachfolgenden Präprozessor-Anweisungen ausgeführt.
- `#ifdef, #ifndef` falls der angegebene Name definiert bzw. nicht definiert ist, werden die nachfolgenden Präprozessor-Anweisungen ausgeführt.
- `#line` führt bei Fehlermeldungen zu einem Sprung auf die angegebenen Zeilennummer.
- `#pragma` veranlaßt den Präprozessor zu einer systemabhängigen Handlung.

E Karlsruher Test

Nicht jedermann eignet sich für so schwierige Dinge wie die elektronische Datenverarbeitung. Um Ihnen die Entscheidung zu erleichtern, ob Sie in die EDV einsteigen oder sich angenehmeren Dingen widmen sollten, haben wir ganz besonders für Sie einen Test entwickelt. Woran denken Sie bei:

Bit	Bier aus der Eifel (1 Punkt)
	Schraubendrehereinsatz (1)
	kleinste Dateneinheit (2 Punkte)
Festplatte	Was zum Essen, vom Partyservice (1)
	Schallplatte (0)
	Massenspeicher (2)
Menü	Was zum Essen (1)
	Dialogtechnik (2)
	mittelalterlicher Tanz (0)
CPU	politische Partei (0)
	Zentralprozessor (2)
	Carnevalsverein (0)
Linker	Linkshänder (0)
	Anhänger einer Linkspartei (1)
	Programm zum Binden von Modulen (2)
IBM	Ich Bin Müde (1)
	International Business Machines (2)
	International Brotherhood of Magicians (1)
Schnittstelle	Verletzung (1)
	Verbindungsstelle zweier EDV-Geräte (2)
	Werkstatt eines Bartscherers (0)
Slot	Steckerleiste im Computer (2)
	einarmiger Bandit (1)
	niederdeutsch für Kamin (0)

Fortran	starker Lebertran (0) Formal Trash Notation (0) Programmiersprache (2)
Mainframe	Frachtkahn auf dem Main (0) Damit wollte FRIDTJOF NANSEN zum Nordpol (0) großer Computer (2)
PC	Plumpsklo (Gravitationstoilette) (1) Personal Computer (2) Power Computing Language (0)
Puffer	Was zum Essen, aus Kartoffeln (1) Was am Eisenbahnwagen (1) Zwischenspeicher (2)
Software	Rohstoff für Softice (0) Programme, Daten und so Zeugs (2) was zum Trinken (0)
Port	was zum Trinken (1) Hafen (1) Steckdose für Peripheriegeräte (2)
Strichcode	maschinell lesbarer Code (2) Geheimsprache im Rotlichtviertel (0) Urliste in der Statistik (0)
Chip	was zum Essen (1) was zum Spielen (1) Halbleiterbaustein (2)
Pointer	Hund (1) starker Whisky (0) Zeiger auf Daten, Adresse (2)
Page	Hotelboy (1) englisch, Seite in einem Buch (1) Untergliederung eines Speichers (2)
Character	was manchen Politikern fehlt (1) Schriftzeichen (2) Wasserfall (0)

Betriebssystem	Konzern (0) betriebsinternes Telefonsystem (0) wichtigstes Programm im Computer (2)
Traktor	Papiereinzugsvorrichtung (2) landwirtschaftliches Fahrzeug (1) Zahl beim Multiplizieren (0)
Treiber	Hilfsperson bei der Jagd (1) Programm zum Ansprechen der Peripherie (2) Vorarbeiter (0)
Animator	was zum Trinken (1) Unterhalter (1) Programm für bewegte Grafik (2)
Hackbrett	Musikinstrument (1) Werkzeug im Hackbau (0) Tastatur (2)
emulieren	nachahmen (2) Öl in Wasser verteilen (0) entpflichten (0)
Font	Menge von Schriftzeichen (2) Soßengrundlage (1) Hintergrund, Geldmenge (0)
Server	Brettsegler (0) Kellner (0) Computer für Dienstleistungen (2)
Yabbawhap	Datenkompressionsprogramm (2) Kriegsruf der Südstadt-Indianer (0) was zum Essen (0)
Terminal	Schnittstelle Mensch - Computer (2) Bahnhof oder Hafen (1) Zubehör zu Drahttauwerk (1)
Ampersand	Sand aus der Amper (1) et-Zeichen, Kaufmanns-Und (2) Untiefe im Wattenmeer (0)

Alias	altgriechisches Epos (0) alttestamentarischer Prophet (0) Zweitname (2)
Buscontroller	Busfahrer (0) Busschaffner (0) Programm zur Steuerung eines Datenbusses (2)
Algol	was zum Trinken (0) Doppelstern (1) Programmiersprache (2)
Rom	Stadt in Italien (1) schwedisch für Rum (1) Read only memory (2)
Dram	Dynamic random access memory (2) dänisch für Schnaps (1) Straßenbahn (0)
Diskette	Mädchen, das oft in Discos geht (0) weiblicher Diskjockey (0) Massenspeicher (2)
Directory	oberste Etage einer Firma (0) Inhaltsverzeichnis (2) Kunststil zur Zeit der Franz. Revolution (0)
Dekrement	was die Verdauung übrig läßt (0) Anordnung von oben (0) Wert, um den ein Zähler verringert wird (2)
Sprungbefehl	Vorkommnis während Ihres Wehrdienstes (0) Kommando im Pferdesport (0) Anweisung in einem Programm (2)
Oktalzahl	Maß für die Klopffestigkeit (0) Zahl zur Basis 8 (2) Anzahl der Oktaven einer Orgel (0)
Subroutine	Kleidungsstück eines Priesters (0) was im Unterbewußten (0) Unterprogramm (2)

Spoiler	Was zum Essen (0) Posting in den Netnews (2) Was am Auto (1)
virtuell	tugendhaft (0) die Augen betreffend (0) nicht wirklich vorhanden, scheinbar (2)
Klammeraffe	ASCII-Zeichen (2) Bürogerät (1) Affenart in Südamerika (0)
ESC	Eisenbahner-Spar- und Creditverein (0) Eishockeyclub (0) escape, Fluchtsymbol (2)
Monitor	Karlsruher Brauerei (0) Fernsehsendung (1) Bildschirmgerät, Überwachungsprogramm (2)
Unix	Tütensuppe (0) Freund von Asterix und Obelix (0) hervorragendes Betriebssystem (2)
Joystick	Computerzubehör (2) männlicher Körperteil (0) Hebel am Spielautomat (0)
Maus	kleines Säugetier (1) Computerzubehör (2) junge Dame (1)
Icon	russisches Heiligenbild (0) Sinnbild (2) Kamerafabrik (0)
Pascal	französischer Mathematiker (1) Maßeinheit für Druck (1) Programmiersprache (2)
Wysiwyg	englisch für Wolpertinger (0) französisch für Elmentritschen (0) what you see is what you get (2)

Register	was in Flensburg (1) was an der Orgel (1) Speicher (2)
Record	was im Sport (1) englisch für Blockflöte (0) Datensatz (2)
HP	High Price (0) Hewlett-Packard (2) Horse Power (1)
Kermit	Klebstoff (0) Frosch aus der Muppet-Show (1) Fileübertragungs-Protokoll (2)
Ethernet	Baustoff (Asbestzement) (0) Local Area Network (2) Student der ETH Zürich (0)
Algorithmus	Übermäßiger Genuß geistiger Getränke (0) Krankheit (0) Rechenvorschrift (2)
File	Was zum Essen (0) Menge von Daten (2) Durchtriebener Kerl (0)
Bug	Vorderteil eines Schiffes (1) Fehler im Programm (2) englisch für Wanze (1)
Router	jemand mit Routine (0) französischer LKW-Fahrer (0) Verbindungsglied zweier Netze (2)
Zylinder	Kopfbedeckung (1) Teil einer Kolbenmaschine (1) Unterteilung eines Plattenspeichers (2)
FTP	kleine, aber liberale Partei (0) File Transfer Protocol (2) Floating Point Processor (0)

Domäne	Geist(0) Bereich (2) Blume (0)
Bridge	Kartenspiel (1) internationales Computernetz (0) Verbindung zweier Computernetze (2)
Email	Glasur (1) elektronische Post (2) Sultanspalast (0)
Baum	was im Wald (Wurzel unten) (1) was auf einem Schiff (keine Wurzel) (1) was aus der Informatik (Wurzel oben) (2)
Internet	Schule mit Schlafgelegenheit (0) Zwischenraum (0) Weltweites Computernetz (2)
Split	UNIX-Kommando (2) kantige Steinchen (0) Stadt in Dalmatien (1)
Mini	Damenoberbekleidung (1) kleiner Computer (2) Frau von Mickey Mouse (0)
Cut	Herrenoberbekleidung (1) Colonia Ulpia Traiana (1) UNIX-Kommando (2)
2B	!2B	Parallelprozessor (0) Assembler-Befehl (0) ein Wort Hamlets (2)
Shell	Filmschauspielerin (Maria S.) (0) Kommando-Interpreter (2) Mineralöl-Gesellschaft (1)
Slip	Unterbekleidung (1) Schlupfschuh (0) Internet-Protokoll (2)

Diäresis	Durchfall (0) Diakritisches Zeichen (Umlaute) (2) Ernährungslehre (0)
Space Bar	Kneipe im Weltraum (www.spacebar.com) (0) Maßeinheit für den Druck im Weltraum (0) Größte Taste auf der Tastatur (2)
Popper	Popcorn-Röster (0) Mail-Programm (2) Philosoph aus Wien (1)
Rohling	Wüster Kerl (1) Noch zu beschreibende CD/DVD (2) Rohkost-Liebhaber (0)
Schleife	Kleidungsstück (1) Schlitterbahn (1) Kontrollanweisung eines Programmes (2)
Alex	Altlasten-Expertensystem (1) Automatic Login Executor (1) Globales Filesystem (1)
Altair	Stern (Alpha Aquilae) (1) Gebirge in Zentralasien (0) früher Personal Computer (2)
Halbbitter	Was zum Essen (Schokolade) (1) Strom- und bitsparender Prozessor (0) Was zum Trinken (0)
Eure Priorität	Anrede des Priors in einem Kloster (0) Anrede des Ersten Sekretärs im Vatikan (0) Anrede des System-Managers (6)

Zählen Sie Ihre Punkte zusammen. Die Auswertung ergibt Folgendes:

- über 170 Punkte: Überlassen Sie das Rechnen künftig dem Computer.
- 85 bis 170 Punkte: Mit etwas Fleiß wird aus Ihnen ein EDV-Experte.
- 18 bis 84 Punkte: Machen Sie eine möglichst steile Karriere außerhalb der EDV und suchen Sie sich fähige Mitarbeiter.
- unter 18 Punkten: Vielleicht hatten Sie schlechte Lehrer?

F GNU Lizenzen

F.1 GNU General Public License

Kopiert von `http://www.gnu.org/copyleft/`.

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) <year> <name of author>
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

F.2 GNU Free Documentation License

Kopiert von `http://www.gnu.org/copyleft/`.

Preamble

The purpose of this license is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this license preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This license is a kind of *copyleft*, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this license in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this license is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this license principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This license applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this license. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **you**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **Modified Version** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **Secondary Section** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **Invariant Sections** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this license. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **Cover Texts** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this license. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **Transparent** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the Document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose

markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not *Transparent* is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **Title Page** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this license requires to appear in the title page. For works in formats which do not have any title page as such, TTitle Pagemmeans the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **Entitled XYZ** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **Acknowledgements**, **Dedications**, **Endorsements**, or **History**.) To **Preserve the Title** of such a section when you modify the Document means that it remains a section *Entitled XYZ* according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this license applies to the Document. These Warranty Disclaimers are considered to be included by reference in this license, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this license.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this license, the copyright notices, and the license notice saying this license applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this license. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general

network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this license, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this license, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this license.

I. Preserve the section entitled *History*, Preserve its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled *History* in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the *History* section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section entitled *Acknowledgements* or *Dedications*, Preserve the title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled *Endorsements*. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be entitled *Endorsements* or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled *Endorsements*, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this license give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this license, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this license, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled *History* in the various original documents, forming one section entitled *History*; likewise combine any sections entitled *Acknowledgements*, and any sections entitled *Dedications*. You must delete all sections entitled *Endorsements*.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this license, and replace the individual copies of this license in the various documents with a single copy that is included in the collection, provided that you follow the rules of this license for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this license, provided you insert a copy of this license into the extracted document, and follow this license in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an

aggregate if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this license does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this license, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this license and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this license or a notice or disclaimer, the original version will prevail.

If a section in the Document is entitled *Acknowledgements*, *Dedications*, or *History*, the requirement (section 4) to Preserve its title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this license. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this license. However, parties who have received copies, or rights, from you under this license will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the license is given a distinguishing version number. If the Document specifies that a particular numbered version of this license *or any later version* applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this license, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this license for your documents

To use this license in a document you have written, include a copy of the license in the Document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the *with...Texts.* line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

G Zeittafel

Ausführlichere Angaben sind den im Anhang **??** *Zum Weiterlesen* in Abschnitt *Geschichte* auf Seiete **??** aufgeführten Werken zu entnehmen. Die meisten Errungenschaften entwickelten sich über manchmal lange Zeitspannen, so dass vor einige Jahreszahlen *um etwa* zu setzen ist.

-10E8	Der beliebte Tyrannosaurus hatte zwei Finger an jeder Hand und rechnete vermutlich im Dualsystem, wenn überhaupt.
-2000	Die Babylonier verwenden für besondere Aufgaben ein gemischtes Stellenwertsystem zur Basis 60.
-400	In China werden Zählstäbchen zum Rechnen verwendet.
20	In der Bergpredigt wird das Binärsystem erwähnt (Matth. 5, 37). Die Römer schieben Rechensteinchen (calculi).
600	Die Inder entwickeln das heute übliche reine Stellenwertsystem, die Null ist jedoch älter. Etwa gleichzeitig entwickeln die Mayas in Mittelamerika ein Stellenwertsystem zur Basis 20.
1200	Leonardo von Pisa, genannt Fibonacci, setzt sich für die Einführung des indisch-arabischen Systems im Abendland ein.
1550	Die europäischen Rechenmeister verwenden sowohl die römische als auch die indisch-arabische Schreibweise.
1617	John Napier erfindet die Rechenknochen (Napier's Bones).
1623	Erste mechanische Rechenmaschine mit Zehnerübertragung und Multiplikation, von Wilhelm Schickard, Tübingen.
1642	Rechenmaschine von Blaise Pascal, Paris für kaufmännische Rechnungen seines Vaters.
1674	Gottfried Wilhelm Leibniz baut eine mechanische Rechenmaschine für die vier Grundrechenarten und befasst sich mit der dualen Darstellung von Zahlen. In der Folgezeit technische Verbesserungen an vielen Stellen in Europa.
1714	Henry Mill erhält ein Patent auf eine Schreibmaschine.
1801	Joseph Marie Jacquard erfindet die Lochkarte und steuert Webstühle damit.
1821	Charles Babbage stellt der Royal Astronomical Society eine programmierbare mechanische Rechenmaschine vor, die jedoch keinen wirtschaftlichen Erfolg hat. Er denkt auch an das Spielen von Schach oder Tic-tac-toe auf Maschinen.
1840	Samuel Finley Breeze Morse entwickelt einen aus zwei Zeichen plus Pausen bestehenden Telegrafencode, der die Buchstaben entsprechend ihrer Häufigkeit codiert.
1847	George Boole entwickelt die symbolische Logik.
1861	Johann Philipp Reis erfindet das Telephon.
1873	Eliphalet Remington and Sons, NY, stellen außer Gewehren und Nähmaschinen auch Schreibmaschinen her. 1886 trennen sie sich vom Schreibmaschinenbereich, der später den Namen Remington Rand und noch später den Namen Sperry Rand trägt.
1876	Alexander Graham Bell erhält ein Patent auf sein Telefon.
1877	Gründung der Bell Telephone Company.
1885	Aus Bell Telephone Co. wird American Telephone + Telegraph Co.
1890	Herman Hollerith erfindet die Lochkartenmaschine und setzt sie bei einer Volkszählung in den USA ein. Das ist der Anfang von IBM.
1894	Otto Luegers *Lexikon der gesamten Technik* führt

	unter dem Stichwort *Elektrizität* als Halbleiter Äther, Alkohol, Holz und Papier auf.
1895	Erste Übertragungen mittels Radio (ALEXANDER POPOW, GUGLIELMO MARCONI).
1896	Gründung der Tabulating Machine Company, der späteren IBM.
1898	VALDEMAR POULSEN erfindet die magnetische Aufzeichnung von Tönen (*Telegraphon*).
1900	01. Januar 1900 00:00:00 GMT Nullpunkt der gegenwärtigen NTP-Ära (eine NTP-Ära umfasst 136 Jahre).
1910	Gründung der Deutschen Hollerith Maschinen GmbH, Berlin, der Vorläuferin der IBM Deutschland.
1918	Das Enigma-Verschlüsselungsverfahren entwickelt.
1924	Aus der Tabulating Machine Company von HERMAN HOLLERITH, später in Computing-Tabulating-Recording Company umbenannt, wird die International Business Machines (IBM).
1930	EDWIN LINK baut – anstatt Pianos und Orgeln wie sein Vater – einen mechanischen Flugsimulator für Übungs- und Vergnügungszwecke und erhält ein Patent darauf. Der Link-Trainer erlangt Verbreitung.
1932	Die Firma AEG baut das erste Tonbandgerät mit dem Namen *Magnetophon*. Die Bänder dazu liefert die BASF.
1937	ALAN TURING veröffentlicht sein Computermodell.
1938	KONRAD ZUSE stellt den programmgesteuerten Rechner Z 1 fertig. Ein späterer Nachbau schafft 1 Rechenoperation pro Sekunde. Elektronische binäre Addiermaschine von JOHN VINCENT ATANASOFF und CLIFFORD BERRY, Iowa State University, zur Lösung linearer Gleichungssysteme.
1939	KONRAD ZUSE stellt die Z 2 fertig. Gründung der Firma Hewlett-Packard, Palo Alto, Kalifornien durch WILLIAM HEWLETT und DAVID PACKARD. Ihr erstes Produkt ist ein Oszillator für Tonfrequenzen (Messtechnik).
1941	KONRAD ZUSE stellt die Z 3 fertig.
1942	Die Purdue University beginnt mit der Halbleiterforschung und untersucht Germaniumkristalle.
1943	Der Computer *Colossus*, Bletchley Park/Buckinghamshire UK, entschlüsselt deutsche Militärnachrichten (Enigma). IBM-Chef THOMAS WATSON schätzt den weltweiten Bedarf an Computern auf 5 (fünf) Stück.
1944	Die Zuse Z 4 wird fertig (2200 Relais, mechanischer Speicher). Sie arbeitet von 1950 bis 1960 in der Schweiz. An der Harvard University bauen HOWARD AIKEN und GRACE HOPPER die Mark I in Relaistechnik. Die Maschine läuft bis 1959.
1945	KONRAD ZUSE entwickelt den Plankalkül, die erste höhere Programmiersprache. WILLIAM BRADFORD SHOCKLEY startet ein Forschungsprojekt zur Halbleiterphysik in den Bell-Labs. VANNEVAR BUSH entwickelt ein System zur Informationsspeicherung und -suche, das auf Mikrofilmen beruht.
1946	JOHN VON NEUMANN veröffentlicht sein Computerkonzept. JOHN PRESPER ECKERT und JOHN WILLIAM MAUCHLY bauen in den USA die ENIAC (Electronic Numerical Integrator and Computer). Die ENIAC rechnet dezimal, enthält 18000 Vakuumröhren, wiegt 30 t, ist 5,5 m hoch und 24 m lang, braucht für eine Addition 0,2 ms, ist an der Entwicklung der Wasserstoffbombe beteiligt und arbeitet bis 1955. Sie ist der Urahne der UNIVAC.
1948	CLAUDE ELWOOD SHANNON begründet die Informationstheorie. JOHN BARDEEN, WALTER HOUSER BRATTAIN und

	WILLIAM BRADFORD SHOCKLEY entwickeln in den Bell-Labs den Transistor, der 10 Jahre später die Vakuumröhre ablöst.
1949	Erster Schachcomputer: Manchester MADM. Das Wort *Bit* kreiert.
1950	An der ETH Zürich geht die Zuse Z 4 in Betrieb.
1952	IBM bringt ihre erste elektronische Datenverarbeitungsanlage, die IBM 701, heraus.
1953	IBM baut die erste Magnetbandmaschine zur Datenspeicherung (726).
1954	Remington-Rand bringt die erste UNIVAC heraus, IBM die 650. Silizium beginnt, das Germanium zu verdrängen.
1955	IBM entwickelt die erste höhere Programmiersprache, die Verbreitung erlangt: FORTRAN (Formula Translator) und verwendet Transistoren in ihren Computern.
1956	KONRAD ZUSE baut die Z 22, die mit Röhren arbeitet. Sie kommt 1958 auf den Markt. Bis 1961 werden 50 Stück verkauft. BARDEEN, BRATTAIN und SHOCKLEY erhalten den Nobelpreis für Physik. IBM stellt die erste Festplatte vor (IBM 350 Disk File für den Computer RAMAC 305), Kapazität 5 MByte, groß wie ein Schrank, Gewicht 1 to, bestehend aus 50 Scheiben zu 24 Zoll, 50.000 US-$.
1957	Die IBM 709 braucht für eine Multiplikation 0,12 ms. Weltweit arbeiten rund 1300 Computer. Seminar von Prof. JOHANNES WEISSINGER über *Programmgesteuerte Rechenmaschinen* im SS 1957 der TH Karlsruhe. KARL STEINBUCH (Firma SEL, später TH Karlsruhe) prägt den Begriff *Informatik*. Erster Satellit (Sputnik, Sowjetunion) kreist um die Erde.
1958	Als eine Reaktion auf den Sputnik gründet das us-amerikanische Verteidigungsministerium (DoD) die Denkfabrik Advanced Research Projects Agency (ARPA), die später das ARPANET aufbaut. MARVIN LEE MINSKY prägt den Begriff *Artificial Intelligence*. Die TH Karlsruhe erhält ihren ersten Computer, eine ZUSE Z 22, finanziert vom Land Baden-Württemberg. Die Maschine verwendet 400 Vakuumröhren und wiegt 1 t. Der Arbeitsspeicher fasst 16 Wörter zu 38 Bits, d. h. 76 Byte. Der Massenspeicher, eine Magnettrommel, fasst rund 40 KByte. Eine Gleitkommaoperation dauert 70 ms. Das System versteht nur Maschinensprache (Freiburger Code) und läuft bis 1972. Im SS 1958 hält Priv.-Doz. KARL NICKEL (Institut für Angew. Mathematik) eine Vorlesung *Programmieren mathematischer und technischer Probleme für die elektronische Rechenmaschine Z 22*. Die Programmiersprache ALGOL 58 kommt heraus. Bei Texas Instruments baut JACK ST. CLAIR KILBY den ersten IC; im Jahr 2000 erhält er dafür den Nobelpreis für Physik.
1959	Im SS 1959 hält Priv.-Doz. KARL NICKEL erstmals die Vorlesung *Programmieren I*, im WS 1959/60 die Vorlesung *Programmieren II*. Erstes Werk von Hewlett-Packard in Deutschland. Siemens baut die Siemens 2002.
1960	Programmieren steht noch in keinem Studienplan, sondern ist freiwillig. Die Karlsruher Z 22 arbeitet Tag und Nacht. Die Programmiersprache COBOL wird veröffentlicht. Ein Computerspiel namens *Spacewar* läuft auf einer Digital Equipment Corporation (DEC) PDP-1 im MIT. ALAN SHUGART entwickelt ein Verfahren zur Aufzeichnung von Daten auf einer magnetisch beschichteten Scheibe.
1961	Die TH Karlsruhe erhält im Zuge der Berufungsverhandlungen von Prof. Nickel eine Zuse Z 23, die mit 2400 Transistoren arbeitet. Ihr Hauptspeicher fasst 240 Wörter zu 40 Bits.

Eine Gleitkommaoperation dauert 15 ms. Außer Maschinensprache versteht sie ALGOL.
Weltweit arbeiten etwa 7300 Computer.

1962 Die TH Karlsruhe erhält einen SEL ER 56, der bis 1968 läuft.
An der Purdue University wird die erste Fakultät für Informatik (Department of Computer Science) gegründet.
Texas Instruments und Fairchild nehmen die Serienproduktion von ICs (Chips) auf.
JOSEPH CARL ROBNETT LICKLIDER hat zwei Visionen: den interaktiven Computer und das galaktische Netz (wenn schon, denn schon). Er wird Direktor in der ARPA und geht an die Verwirklichung seiner Visionen.

1963 Weltweit arbeiten etwa 16.500 Computer.
Erster geostationärer Satellit (Syncom).
IVAN E. SUTHERLAND entwickelt in seiner Doktorarbeit am MIT das Sketchpad, einen grafischen Bildschirm mit Lichtgriffel, und wird damit zum Vater der Computergrafik.

1964 Die Programmiersprache BASIC erscheint.
DOUGLAS CARL ENGELBART erfindet am Stanford Research Institute die Maus und die Fenstertechnik.
IBM legt das Byte zu 8 Bits fest (IBM 360).
Ein Chip enthält auf 0,5 cm^2 10 Transistoren.

1965 Beginn des Betriebssystems MULTICS bei MIT, Bell und General Electric. Aus dessen Misserfolg erwächst später UNIX.

1966 Die TH Karlsruhe erhält eine Electrologica X 8, die bis 1973 betrieben wird. Gründung des Karlsruher Rechenzentrums.
Hewlett-Packard steigt in die Computerei ein (HP 2116 A).

1967 Erster elektronischer Taschenrechner (Texas Instruments).
IVAN E. SUTHERLAND entwickelt an der Harvard University einen Helm mit binokularem Display und bringt damit die Virtual Reality ein gutes Stück voran.

1968 Am 26. Februar entscheiden sich maßgebende Vertreter der Computerwissenschaft im deutschsprachigen Raum im Anschluss an ein internationales Kolloquium in Dresden für die Bezeichnung *Informatik* nach französischem Beispiel.
Die Programmiersprache PASCAL kommt heraus. Die Firma Intel gegründet. Hewlett-Packard baut den ersten wissenschaftlichen programmierbaren Tischrechner (HP 9100 A).

1969 In Karlsruhe wird am 1. Januar das Institut für Informatik gegründet, Direktor KARL NICKEL. Im WS 1969/70 beginnt in Karlsruhe die Informatik als Vollstudium mit 91 Erstsemestern.
Gründung der Gesellschaft für Informatik (GI) in Bonn.
In den Bell Labs UNIX in Assembler auf einer DEC PDP 7.
Beginn des ARPANET-Projektes, erste Teilnehmer U. of California at Los Angeles, Stanford Research Institute, U. of California at Santa Barbara und U. of Utah, allesamt mit DEC PDP-10 Maschinen.
RFC 0001: Host Software, von STEVE CROCKER.

1970 Die Universität Karlsruhe erhält eine UNIVAC 1108, die bis 1987 läuft und damit den hiesigen Rekord an Betriebsjahren hält. Preis 23 MDM, 3 Zentraleinheiten, 256 Kilo-Wörter zu je 36 Bits Arbeitsspeicher, 20 Bildschirme.
Die Karlsruher Fakultät für Informatik wird gegründet.
Am 01. Januar 1970 00:00:00 GMT beginnt die UNIX-Uhr zu laufen.

1971 UNIX auf C umgeschrieben, erster Mikroprozessor (Intel 4004).
ALAN SHUGART entwickelt bei IBM die Floppy Disk.
Die Internet-Protokolle FTP (RFC 114) und Telnet (RFC 137) werden vorgeschlagen und diskutiert.

1972 IBM entwickelt das Konzept des virtuellen Speichers und

	stellt die 8-Zoll-Floppy-Disk vor. Xerox (ROBERT METCALFE), DEC und Intel entwickeln den Ethernet-Standard.
	Das ARPANET wird der Öffentlichkeit vorgestellt.
	Ein Student namens STEPHAN G. WOZNIAK lötet sich einen Computer zusammen, der den Smoke-Test nicht übersteht.
	In der Bundesrepublik arbeiten rund 8.200 Computer.
	Erster wissenschaftlicher Taschenrechner (Hewlett-Packard 35).
1973	Erste internationale Teilnehmer am ARPANET: NORSAR (Norwegian Seismic Array), Norwegen und U. College of London.
1974	Der erste programmierbare Taschenrechner kommt auf den Markt (Hewlett-Packard 65), Preis 2500 DM.
1975	UNIX wird veröffentlicht (Version 6), Beginn der BSD-Entwicklung.
	Die Zeitschrift *Byte* wird gegründet.
	Erste, mäßig erfolgreiche Personal Computer (Xerox, Altair).
	Die Firma Microsoft Corporation von WILLIAM HENRY GATES III. und PAUL ALLEN gegründet.
1976	STEVEN P. JOBS und STEPHAN G. WOZNIAK gründen die Firma Apple und bauen den Apple I. Er kostet 666,66 Dollar.
	ALAN SHUGART stellt die 5,25-Zoll-Diskette vor.
	Die nichtprozedurale Datenbanksprache SQL – entwickelt von EDGAR F. CODD bei IBM – wird veröffentlicht.
	WHITFIELD DIFFIE und MARTIN E. HELLMANN veröffentlichen die erste Arbeit über unsymmetrische Verschlüsselung.
1977	ROBERT E. KAHN und VINTON GRAY CERF veröffentlichen das Konzept von TCP/IP, anfangs Kahn-Cerf-Protokolle genannt.
1978	In der Bundesrepublik arbeiten rund 170.000 Computer.
	Der Commodore PET 2001 – ein Vorläufer des C64 – kommt heraus, 4 bis 32 kbyte Arbeitsspeicher, Bildschirm 25 Zeilen zu 40 Zeichen.
	Erste Tabellenkalkulation *Visicalc*, für den Apple II, von DAN BRICKLIN und BOB FRANKSTON, Harvard University.
	Erste Fassung von TEX (DONALD ERVIN KNUTH) veröffentlicht.
	Das Network Time Protocol (NTP) wird in Gebrauch genommen.
1979	Faxdienst in Deutschland eingeführt.
	Beginn des Usenet in der Duke University und der University of North Carolina auf der Basis von uucp-Verbindungen.
	Die Zusammenarbeit von Apple mit Rank Xerox führt zur Apple Lisa, ein Mißerfolg, aber der Wegbereiter für den Macintosh.
	Plattenherstellerfirma *Seagate* gegründet.
	Gründung der Satelliten-Kommunikations-Firma Inmarsat.
	BJARNE STROUSTRUP beginnt mit der Entwicklung von C++.
	Programmiersprache Ada veröffentlicht.
	Betriebssystem DOS für Intel 8086/8088 von Fa. Seattle Computer Products entwickelt, später von Microsoft erworben.
1980	Erster Jugendprogrammier-Wettbewerb der GI.
	Erster Home-Computer: Sinclair ZX-80, für rund 500 DM.
	Sony führt die 3,5-Zoll-Diskette ein. In den Folgejahren entwickeln andere Firmen auch Disketten mit Durchmessern von 3 bis 4 Zoll.
	Microsoft bringt Xenix, ein UNIX für PCs, heraus.
1981	Die Universität Karlsruhe erhält eine Siemens 7881 als zentralen Rechner.
	IBM bringt in den USA den IBM-PC heraus mit PC-DOS 1.0 (MS DOS) als wichtigstem Betriebssystem.
	In Berlin wird der *Chaos Computer Club* gegründet.
	Xanadu-Projekt von TED NELSON, ein Vorläufer des Web.
1982	Die Firma Sun Microsystems wird gegründet, entscheidet sich für UNIX und baut die ersten Workstations.
	JIM CLARK gründet Silicon Graphics, Inc. (SGI)

	Beginn des EuNETs, einer der ersten deutschen Internet-Provider, an der Universität Dortmund.
	WILLIAM GIBSON prägt das Wort *Cyberspace*.
	MORTON HEILIG präsentiert einen Spielautomaten für Motorrad- und Auto-Simulationen mit Stereotonfilm, Gebläse, Gerüchen und vibrierenden Sitzen, echt multimedial, aber erfolglos, da zu teuer.
1983	Die Universität Karlsruhe erhält einen Vektorrechner Cyber 205 und eine Siemens 7865. Die Cyber leistet 400 Mio. Flops.
	Beginn des *Lokalen Informatiknetzes Karlsruhe* (LINK), ab 1984 Xlink, in der Fakultät für Informatik der Universität Karlsruhe.
	IBM bringt den PC auf den deutschen Markt.
	UNIX kommt als System V von AT&T in den Handel,
	die erste Ausgabe der Zeitschrift *Computertechnik* (c't) erscheint (Nr. 12/83 vom Oktober 1983).
	Gründung der X/Open-Gruppe.
	MS-DOS 2.0 (PC-DOS 2.0) und Novell Netware kommen heraus.
	Microsoft Windows wird angekündigt.
	Das ARPAnet wechselt von NCP auf TCP/IP.
1984	Der erste Apple Macintosh (128K) und der Hewlett-Packard Thinkjet, der erste Tintenstrahldrucker, kommen auf den Markt.
	GNU-Projekt von RICHARD MATTHEW STALLMAN gegründet.
	Der IBM PC/AT mit Prozessor Intel 80 286 und MS-DOS 3.0 kommen heraus.
	Siemens steigt in UNIX (Sinix) ein.
	Die Universität Karlsruhe wird Email-Relay für Deutschland zum Computer Science Net (CSNet) in den USA. Als erste Mail erhält Prof. ZORN, U. Karlsruhe, einen Gruß vom CSNet.
	Entwicklung des X Window Systems am MIT.
1985	MS-Windows 1.0, IBM 3090 und IBM Token Ring Netz.
	XLink an der Universität Karlsruhe stellt als erstes deutsches Netz eine Verbindung zum nordamerikanischen ARPANET her.
	Hewlett-Packard bringt den ersten Laserjet-Drucker heraus.
1986	Weltweit etwa eine halbe Million UNIX-Systeme und 3000 öffentliche Datenbanken.
	Mit dem Computer-Investitionsprogramm des Bundes und der Länder (CIP) kommen mehrere HP 9000/550 unter UNIX an die Universität Karlsruhe.
1987	Microsoft XENIX (ein UNIX) für den IBM PC/AT
	IBM bringt die PS/2-Reihe unter MS-OS/2 heraus.
	Weltweit mehr als 5 Millionen Apple Computer und etwa 100 Millionen PCs nach Vorbild von IBM.
	Das MIT veröffentlicht das X Window System Version 11 (X11).
	In Berkeley wird die RAID-Technologie entwickelt.
	Beginn des baden-württembergischen BelWue-Netzes.
1988	JARKKO OIKARINEN, Finnland, entwickelt den IRC.
	Das Karlsruher Campusnetz KARLA wird durch das Glasfasernetz KLICK ersetzt. VBN-Strecke Karlsruhe - Stuttgart im BelWue-Netz.
	Frankreich geht ans Internet (INRIA, Rocquencourt bei Paris).
	Gründung der Open Software Foundation (OSF) und der UNIX International Inc. MS-DOS 4.0 für PCs.
	Ein Internet-Wurm namens Morris geht auf die Reise, darauf hin Gründung des Computer Emergency Response Teams (CERT).
	Erster Hoax (2400-baud-Modem-Hoax) im Internet, siehe CIAC.
	Erstes landmobiles Satellitensystem für Datenfunk (Inmarsat-C).
1989	Das NFSNET löst das ARPANET als Backbone des Internet ab.
	UNIX System V Release 4 vereinheitlicht System V, BSD und Xenix.
	Im Rechenzentrum Karlsruhe löst die IBM 3090 die

	Siemens 7881 ab. ISDN in Deutschland eingeführt.
	Erster deutscher Internet-Direktanschluss via Xlink, Karlsruhe.
1990	Zunehmende Vernetzung, Anschluss an weltweite Netze.
	Die Internet Society (ISOC) schätzt das Internet auf 500.000 Knoten.
	Computer-Kommunikation mittels E-Mail, Btx und Fax vom Arbeitsplatz aus. Optische Speichermedien (CD-ROM, WORM).
	Das Web (URL, HTTP, HTML) von Timothy Berners-Lee und Robert Cailliau am CERN in Genf entwickelt.
	UNIX System V Version 4.
	Die mittlere Computerdichte in technisch orientierten Instituten und Familien erreicht 1 pro Mitglied.
1991	Das UNIX-System OSF/1 mit dem Mach-Kernel der Carnegie-Mellon-Universität kommt heraus.
	17. Sep.: Anfang von LINUX (Linus Benedict Torvalds).
	Erster Web-Server in den USA: Stanford Linear Accelerator Center.
	MS-DOS 5.0 für PCs. Anfänge von Microsoft Windows NT.
	Das DE-NIC an der Universität Dortmund gegründet.
	IBM, Apple und Motorola kooperieren mit dem Ziel, einen Power PC zu entwickeln.
1992	Die Universität Karlsruhe nimmt den massiv parallelen Computer MasPar 1216A mit 16000 Prozessoren in Betrieb.
	Novell übernimmt von AT&T die UNIX-Aktivitäten (USL).
	FORTRAN 90 verabschiedet.
	Eine Million Knoten im Internet. Weltweit etwa 50 Web-Server.
	Erster deutscher Web-Server, am DESY in Hamburg.
1993	MS-DOS Version 6.0. Microsoft kündigt Windows-NT an.
	DEC stellt PC mit Alpha-Prozessor vor, 150 MHz, 14.000 DM.
	Novell tritt das Warenzeichen UNIX an die X/Open-Gruppe ab.
	Marc Andreessen, NCSA, schreibt einen Web-Browser für das X Window System mit der Möglichkeit, farbige Grafiken darzustellen.
	Patrick Volkerding stellt die Linux-Distribution Slackware zusammen, die erste Distribution von einiger Verbreitung.
	Ian Murdock, Student an der Purdue University, stellt am 16. August das Debian-Projekt vor. Gegen Jahresende Debian GNU/Linux Version 0.90.
	Weltweit etwa 250 Web-Server.
	Das DE-NIC zieht ans Rechenzentrum der Universität Karlsruhe.
1994	Weltweit 10 Mio. installierte UNIX-Systeme prognostiziert.
	Linux 1.0 veröffentlicht.
	Das Internet umfasst etwa 4 Mio. Knoten und 20 Mio. Benutzer.
	Erste Spam-Mail (Canter + Siegel). Erste Banner-Werbung (Wired).
	Marc Andreessen und Jim Clark gründen die Firma Netscape.
1995	Kommerzielle Netze lösen in den USA das NFSNET als Backbone ab.
	Die X/Open-Gruppe führt die Bezeichnung *UNIX 95* für Systeme ein, die der *Single UNIX Specification* genügen.
	Die Universität Karlsruhe ermöglicht in Zusammenarbeit mit dem Oberschulamt nordbadischen Schulen den Zugang zum Internet. Ähnliche Projekte werden auch an einigen anderen Hoch- und Fachhochschulen durchgeführt.
	Die Programmiersprache JAVA wird von Sun veröffentlicht.
	Online-Auktionshaus *Ebay* als Sammlerbörse in den USA gegründet.
	Weltweit etwa 50000 Web-Server.
1996	Die Massen und Medien entdecken das Internet.
	Debian GNU/Linux Version 1.1 (*buzz*) wird veröffentlicht, zum Jahresende folgt Version 1.2 (rex).
	FORTRAN 95, eine revidierte Fassung von FORTRAN 90, fertig.
	Die Open Software Foundation (OSF) und X/Open schließen sich zur Open Group zusammen.

1997 100-Ethernet ist erschwinglich geworden, über das Gigabit-Ethernet wird geredet. In Deutschland gibt es rund 20 Mio. PCs und 1 Mio. Internetanschlüsse (Quelle: Fachverband Informationstechnik).
Debian GNU/Linux Version 1.3 (*bo*) freigegeben, rund 1000 Pakete.
Single UNIX Specification Version 2 im Web veröffentlicht.
HTML 4.0 freigegeben.
Der Buchversender Amazon meldet ein Patent an dergestalt, dass man mit einem Mausklick im Internet eine Ware bestellt.

1998 Compaq übernimmt die Digital Equipment Corporation (DEC).
IBM bringt DOS 2000 heraus, Microsoft kündigt Windows 2000 an.
Debian GNU/Linux Version 2.0 (*hamm*) freigegeben, 1500 Pakete.
KDE 1.0 veröffentlicht. 9-GB-Festplatten kosten 500 DM.
Gigabit-Ethernet-Standard IEEE 802.3z verabschiedet.
JONATHAN B. POSTEL, einer der Apostel des Internet und Autor vieler RFCs, gestorben. Siehe RFC 2441: *Working with Jon* und RFC 2468: *I Remember IANA*.

1999 Das Y2K-Problem – die Jahrtausendwende – beschäftigt die Gemüter, weil die Programmierer früherer Jahrzehnte mit den Bits knauserten.
Der RFC 2550 löst auch gleich das Y10K-Problem.
Debian GNU/Linux Version 2.1 (*slink*) kommt heraus.
Betreiber großer Suchmaschinen schätzen die Anzahl der Web-Seiten weltweit auf 1 Milliarde.
LINUS B. TORVALDS wird Ehrendoktor der Universität Stockholm.

2000 Das Y2K-Problem hat sich praktisch nicht ausgewirkt.
Den 29. Februar 2000 haben wir auch gut überstanden, einen Schalttag nach einer Regel, die nur alle 400 Jahre angewendet wird.
Debian GNU/Linux Version 2.2 (*potato*) kommt heraus, 6500 Pakete.
Microsoft Windows 2000 ist erhältlich. Ein Macro-Virus namens *Love Letter* sorgt für Aufregung – außerhalb der Linux/UNIX-Welt.
Der Intel Pentium kommt bei einer Taktfrequenz von 1,5 GHz an.
Zum Jahresende 2 Mio. Internet-Hosts in Deutschland (Quelle: RIPE).

2001 CLAUDE ELWOOD SHANNON gestorben, gilt als Erfinder des Bits und Begründer der Informationstheorie.
In den USA starten JIMMY DONAL WALES und LAWRENCE MARK SANGER die Wikipedia, eine internationale Online-Enzyklopädie auf der Grundlage eines Wikis.

2002 Die Einführung der Euro-Währung führt zu einem neuen Zeichen in internationalen Zeichensätzen.
Im Herbst befällt der Wurm *Slapper* zahlreiche Webserver unter Linux.
Debian GNU/Linux Version 3.0 (*woody*) wird als stabil freigegeben.
Die Distribution umfasst 8700 Pakete.

2004 PCs werden zunehmend ohne Floppy-Laufwerk ausgeliefert.
Politische Überlegungen, die Patentierung von Software zuzulassen, bedrohen die Open Source Welt.

2005 Anfang Juni wird Debian GNU/Linux Version 3.1 (*sarge*) nach reiflicher Überlegung als stabil freigegeben und löst 3.0r6 (*woody*) ab; die Distribution umfasst rund 15.000 Pakete. *Woody* wird damit zur Old-stable-Version, *etch* rückt zur Testing-Version auf.
In Frankfurt (M) findet die Wikimania 2005 statt, die erste internationale Wikimedia-Konferenz.
Die deutsche Wikipedia nähert sich der Marke von 300.000 Einträgen.
Internet-Telefonie (Voice over IP) ist stark im Kommen.
Prof. KARL STEINBUCH gestorben.
Ende des Jahres kommt X11R7 heraus.
Der weltweit leistungsfähigste Rechner – ein IBM Blue Gene – läuft unter einem Linux (Quelle: `http://www.top500.org/`).

2006 Die englische Wikipedia überschreitet die Marke von 1 Mio. Einträgen,

die deutsche die 400.000, die französische die 300.000.
Laut `http://www.netcraft.com/` sind die am häufigsten besuchten Websites Google, Yahoo, Microsoft, BBC, CNN, Ebay, Fox News, Amazon und Wikipedia.
Das Deutsche Forschungsnetz nimmt das X-Win in Betrieb, eine Netzinfrastruktur mit Anschlüssen bis zu 10 Gigabit/s und 43 Kernstandorten.
Mitte des Jahres gibt es 10 Mio. Einträge unter der Top Level Domain .de, sie ist damit weltweit nach .com die zweitumfangreichste.
Debian stellt den Sicherheitssupport für *woody* ein und erklärt zum Jahresende die Version 4.0 (*etch*) mit Kern 2.6.17 als stabil.

H Zum Weiterlesen

Die Auswahl ist subjektiv und enthält Werke, die wir noch lesen wollen, schon gelesen haben oder sogar oft benutzen.

1. Lexika, Glossare, Wörterbücher

   – Newsgruppen:
     news.answers
     de.etc.lists
     news.lists

   – RFC 1392 (FYI 18): Internet Users' Glossary
     `ftp://ftp.nic.de/pub/rfc/rfc1392.txt`
     1993, 53 S.

   – Duden Informatik
     Dudenverlag, Mannheim, 1993, 800 S.
     Nachschlagewerk, sorgfältig gemacht, theorielastig,
     Begriffe wie Ethernet, LAN, SQL, Internet fehlen.

   – Fachausdrücke der Informationsverarbeitung Englisch – Deutsch, Deutsch – Englisch
     IBM Deutschland, Form-Nr. Q12-1044, 1698 S.
     Wörterbuch und Glossar

   – IBM Terminology
     `http://www-3.ibm.com/ibm/terminology/`

   W. Alex Abkürzungs-Liste ABKLEX (Informatik, Telekommunikation)
     `http://www.ciw.uni-karlsruhe.de/abklex.html`
     `http://www.ciw.uni-karlsruhe.de/abklex.pdf`
     Rund 9000 Abkürzungen aus Informatik und Telekommunikation

   M. Broy, O. Spaniol Lexikon Informatik und Kommunikationstechnik
     Springer, Berlin + Heidelberg, 1999, 863 S.

   E. Kajan Information Technology Encyclopedia and Acronyms
     Springer, Berlin + Heidelberg, 2002, 720 S.

   E. S. Raymond The New Hacker's Dictionary
     The MIT Press, Cambridge, 1996, 547 S.
     Siehe auch `http://www.ciw.uni-karlsruhe.de/kopien/jargon/`
     Begriffe aus dem Netz, die nicht im Duden stehen

2. Informatik

   – Newsgruppen:
     comp.* (alles, was mit Computer Science zu tun hat, mehrere hundert Untergruppen)
     de.comp.* (dito, deutschsprachig)
     alt.comp.*

   W. Coy Aufbau und Arbeitsweise von Rechenanlagen
     Vieweg, Braunschweig, 1992, 367 S.
     Digitale Schaltungen, Rechnerarchitektur, Betriebssysteme am Beispiel von UNIX

   T. Flik, H. Liebig Mikroprozessortechnik
     Springer, Berlin + Heidelberg, 1998, 585 S.
     CISC, RISC, Systemaufbau, Assembler und C

W. K. Giloi Rechnerarchitektur
Springer, Berlin + Heidelberg, 1999, 488 S.

G. Goos Vorlesungen über Informatik
Band 1: Grundlagen und funktionales Programmieren,
Springer, Berlin + Heidelberg, 1997, 394 S.
Band 2: Objektorientiertes Programmieren und Algorithmen,
Springer, Berlin + Heidelberg, 1999, 396 S.
Band 3: Berechenbarkeit, formale Sprachen, Spezifikationen,
Springer, Berlin + Heidelberg, 1997, 284 S.
Band 4: Paralleles Rechnen und nicht-analytische Lösungsverfahren,
Springer, Berlin + Heidelberg, 1998, 292 S.
`i44www.info.uni-karlsruhe.de/~i44www/goos-buch.html`

D. E. Knuth The Art of Computer Programming, 3 Bände
Addison-Wesley, Boston,
Klassiker, stellenweise mathematisch, 7 Bände geplant,
Band 4 soll 2004 fertig sein, Band 5 im Jahr 2009, Homepage
des Meisters: `www-cs-staff.stanford.edu/~uno/index.html`

W. Schiffmann, R. Schmitz Technische Informatik
Springer, Berlin + Heidelberg, 1993/94, 1. Teil Grundlagen der
digitalen Elektronik, 282 S.; 2. Teil Grundlagen der
Computertechnik, 283 S.

K. W. Wagner Einführung in die Theoretische Informatik
Springer, Berlin + Heidelberg, 1994, 238 S.
Grundlagen, Berechenbarkeit, Komplexität, BOOLEsche
Funktionen, Automaten, Grammatiken, Formale Sprachen

3. Algorithmen, Numerische Mathematik

– Newsgruppen:
sci.math.*

J. L. Bentley Programming Pearls
Addison-Wesley, Boston, 1999, 256 S.
Pfiffige Algorithmen und Programmierideen

G. Engeln-Müllges, F. Reutter Formelsammlung zur
Numerischen Mathematik mit C-Programmen
BI-Wissenschaftsverlag, Mannheim, 1990, 744 S.
Algorithmen und Formeln der Numerischen Mathematik
samt C-Programmen.

G. Engeln-Müllges, F. Uhlig Numerical Algorithms with C
Springer, Berlin + Heidelberg, 1996, 596 S.

D. E. Knuth Algorithmen
(deutsche Übersetzung von *Fundamental Algorithms*)
Springer, Berlin + Heidelberg, 2004, 700 S.

K. Loudon Mastering Algorithms in C
O'Reilly, Sebastopol, 1999, 560 S.

T. Ottmann, P. Widmayer Algorithmen und Datenstrukturen
BI-Wissenschafts-Verlag, Mannheim, 1993, 755 S.

W. H. Press u. a. Numerical Recipes in C
Cambridge University Press, 1993, 994 S.

H. R. Schwarz Numerische Mathematik
Teubner, Stuttgart, 1993, 575 S.

R. Sedgewick Algorithmen in C
Addison-Wesley, Bonn, 1992, 742 S.
Erklärung gebräuchlicher Algorithmen und Umsetzung in C

R. Sedgewick Algorithmen in C++
Addison-Wesley, Bonn, 1992, 742 S.

J. Stoer, R. Bulirsch Numerische Mathematik
Springer, Berlin + Heidelberg, 1. Teil 1999, 378 S.,
2. Teil 2000, 375 S.

4. Betriebssysteme

- Newsgruppen:
  comp.os.*
  de.comp.os.*

L. Bic, A. C. Shaw Betriebssysteme
Hanser, München, 1990, 420 S.
Allgemeiner als Tanenbaum + Woodhull

A. S. Tanenbaum, A. S. Woodhull Operating Systems, Design and Implementation
Prentice-Hall, London, 1997, 939 S.
Einführung in Betriebssysteme am Beispiel von UNIX

A. S. Tanenbaum Modern Operating Systems
Prentice-Hall, London, 1992, 728 S.
Allgemeiner und moderner als vorstehendes Buch;
erläutert MS-DOS, UNIX, MACH und Amoeba

A. S. Tanenbaum Distributed Operating Systems
Prentice-Hall, London, 1994, 648 S.

H. Wettstein Systemarchitektur
Hanser, München, 1993, 514 S.
Grundlagen, kein bestimmtes Betriebssystem

5. Linux/UNIX allgemein

- Newsgruppen:
  comp.unix.*
  comp.sources.unix
  comp.std.unix
  de.comp.os.unix
  alt.unix.wizards

M. J. Bach Design of the UNIX Operating System
Prentice-Hall, London, 1987, 512 S.
Dateisystem und Prozesse, wenig zur Shell

S. R. Bourne Das UNIX System V (The UNIX V Environment)
Addison-Wesley, Bonn, 1988, 464 S.
Einführung in UNIX und die Bourne-Shell

P. H. Ganten, W. Alex Debian GNU/Linux
Springer, Berlin + Heidelberg, 2004, 970 S.
Einrichtung, Konfiguration und Betrieb von Debian GNU/Linux

J. Gulbins, K. Obermayr, Snoopy Linux
Springer, Berlin + Heidelberg, 2003, 900 S.
Benutzung von Linux/UNIX, geht in Einzelheiten der Kommandos

H. Hahn A Student's Guide to UNIX
McGraw-Hill, New York, 1993, 633 S.
Einführendes Lehrbuch, mit Internet-Diensten

B. W. Kernighan, R. Pike Der UNIX-Werkzeugkasten
Hanser, München, 1986, 402 S.
Gebrauch vieler UNIX-Kommandos

M. Kofler Linux – Installation, Konfiguration, Anwendung
Addison-Wesley, Bonn, 2000, 1108 S.
5. Auflage, spricht für das Buch.

D. G. Korn, M. I. Bolsky The Kornshell, Command and Programming Language
deutsch: Die KornShell, Hanser, München, 1991
Einführung in UNIX und die Korn-Shell

A. Robbins UNIX in a Nutshell
O'Reilly, Sebastopol, 2000, 632 S.
Nachschlagewerk zu den meisten UNIX-Kommandos, im UNIX CD Bookshelf enthalten. Auch auf Englisch.

M. J. Rochkind Advanced UNIX Programming
Addison-Wesley, Boston, 2004, 719 S.
Beschreibung der wichtigsten UNIX System Calls

K. Rosen u. a. UNIX: The Complete Reference
Osborne/McGraw-Hill, Berkeley, 1999, 1302 S.
Fast würfelförmiges Nachschlagewerk, insbesondere zu Linux, Solaris und HP-UX; breites Themenspektrum

E. Siever et al. LINUX in a Nutshell
O'Reilly, Sebastopol, 2001, 880 S.
Nachschlagewerk zu den meisten LINUX-Kommandos

W. R. Stevens Advanced Programming in the UNIX Environment
Addison-Wesley, Boston, 1992, 744 S.
Ähnlich wie Rochkind

6. Linux/UNIX Verwaltung

Æ. Frisch Essential System Administration
O'Reilly, Sebastopol, 1995, 760 S.
Übersicht für Benutzer auf dem Weg zum Sysadmin.

K. Heuer, R. Sippel UNIX-Systemadministration
Springer, Berlin + Heidelberg, 2004, 800 S.

E. Nemeth, G. Snyder, S. Seebass, T. R. Hein UNIX System Administration Handbook
Prentice-Hall, Englewood-Cliffs, 2001, 835 S.
Auf den neuesten Stand gebrachte Hilfe für Sysadmins, viel Stoff.

R. U. Rehman HP Certified – HP-UX System Administration
Prentice Hall PTR, Upper Saddle River, 2000, 800 S.
Begleitbuch zu einem Kurs, Einführung in und Verwaltung von HP-UX

M. Welsh, M. K. Dalheimer, L. Kaufmann Running Linux
O'Reilly, Sebastopol, 1999, 750 S.
Einrichtung und Betrieb eines LINUX-PCs

7. Linux/UNIX Einzelthemen

– Newsgruppen:
comp.unix.*

A. V. Aho, B. W. Kernighan, P. J. Weinberger The AWK Programming Language
Addison-Wesley, Boston, 1988, 210 S.
Standardwerk zum AWK

D. Cameron, B. Rosenblatt Learning GNU Emacs
O'Reilly, Sebastopol, 1991, 442 S.

D. Dougherty, A. Robbins sed & awk
O'Reilly, Sebastopol, 1997, 407 S.

H. Herold Linux Unix Profitools: awk, sed, lex, yacc und make
Addison-Wesley, München, 1998, 890 S.

L. Lamb, A. Robbins Textbearbeitung mit dem vi-Editor
O'Reilly, Köln, 1999, 333 S.

A. Oram, S. Talbott Managing Projects with make
O'Reilly, Sebastopol, 1993, 149 S.

L. Wall, T. Christiansen, J. Orwant Programming Perl
O'Reilly, Sebastopol, 2000, 1067 S.

8. X Window System (X11), Motif, Gnome, KDE

- Newsgruppen:
  comp.windows.x.*

- OSF/Motif Users's Guide
  OSF/Motif Programmer's Guide
  OSF/Motif Programmer's Reference
  Prentice-Hall, Englewood Cliffs, 1990

F. Culwin An X/Motif Programmer's Primer
Prentice-Hall, New York, 1994, 344 S.

T. + M. K. Dalheimer KDE Anwendung und Programmierung
O'Reilly, Sebastopol, 1999, 321 S.

K. Gottheil u. a. X und Motif
Springer, Berlin + Heidelberg, 1992, 694 S.

N. Mansfield The Joy of X
Addison-Wesley, Boston, 1993, 368 S.
Als Einstieg für Anwender geeignet.

A. Nye XLib Programming Manual
O'Reilly, Sebastopol, 1990, 635 S.
Einführung in X11 und den Gebrauch der XLib

V. Quercia, T. O'Reilly X Window System Users Guide
O'Reilly, Sebastopol, 1990, 749 S.
Einführung in X11 für Anwender

R. J. Rost X and Motif Quick Reference Guide
Digital Press, Bedford, 1993, 400 S.

9. Textverarbeitung mit LaTeX

K. Braune, J. Lammarsch, M. Lammarsch LaTeX
Springer, Berlin + Heidelberg, 2004, 700 S.

M. K. Dalheimer LaTeX kurz & gut
O'Reilly, Köln, 2000, 72 S.

H. Kopka LaTeX, 3 Bände
Band 1: Einführung
Addison-Wesley, Bonn, 2000, 520 S.
Band 2: Ergänzungen
Addison-Wesley, Bonn, 1997, 456 S.
Band 3: Erweiterungen
Addison-Wesley, Bonn, 1996, 512 S.
Standardwerk im deutschen Sprachraum

L. Lamport Das LaTeX-Handbuch
Addison-Wesley, Bonn, 1995, 360 S.

H. Partl u. a. LaTeX-Kurzbeschreibung
ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/latex/lkurz.ps.gz
ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/latex/lkurz.tar.gz
1990, 46 S., Postscript und LaTeX-Quellen
Einführung, mit deutschsprachigen Besonderheiten (Umlaute)

10. Multimedia (Grafik, Sound)

– Newsgruppen:
comp.graphics.*
alt.graphics.*

J. D. Foley Computer Graphics – Principles and Practice
Addison-Wesley, Boston, 1992, 1200 S.
Standardwerk zur Computer-Raster-Grafik

R. F. Ferraro Programmer's Guide to the EGA and VGA Cards
Addison-Wesley, Boston, 1990, 1040 S.
Viele Grundlagen, die über EGA und VGA hinausgehen

K. Kylander, O. S. Kylander GIMP
MITP-Verlag, Bonn, 1999, 700 S.
Benutzerhandbuch zum GNU Image Manipulation Program

11. Programmieren allgemein

– Newsgruppen:
comp.programming
comp.unix.programmer
comp.lang.*
comp.software.*
comp.software-eng
comp.compilers
de.comp.lang.*

A. V. Aho u. a. Compilers, Principles, Techniques and Tools
Addison-Wesley, Boston, 1986, 796 S.

B. Beizer Software Testing Techniques
Van Nostrand-Reinhold, 1990, 503 S.

F. P. Brooks jr. The Mythical Man-Month
Addison-Wesley, Boston, 1995, 322 S.
Organisation großer Software-Projekte

M. K. Dalheimer Linux – Wegweiser zu Programmierung + Entwicklung
O'Reilly, Sebastopol, 1997, 580 S.
Software-Entwicklung unter LINUX, Werkzeuge

N. Ford Programmer's Guide
ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/misc/pguide.txt
1989, 31 S., ASCII
allgemeine Programmierhinweise, Shareware-Konzept

T. Grams Denkfallen und Programmierfehler
Springer, Berlin + Heidelberg, 1990, 159 S.
PASCAL-Beispiele, gelten aber auch für C-Programme

D. Gries The Science of Programming
Springer, Berlin + Heidelberg, 1981, 366 S.
Grundsätzliches zu Programmen und ihrer Prüfung,
mit praktischer Bedeutung.

R. H. Güting, M. Erwig Übersetzerbau
Springer, Berlin + Heidelberg, 1999, 368 S.

M. Marcotty, H. Ledgard The World of Programming Languages
Springer, Berlin + Heidelberg, 1987, 360 S.

S. Pfleeger Software Engineering: The Production of Quality Software
Macmillan, 1991, 480 S.

I. W. Ricketts Managing Your Software Project – A Student's Guide
Springer, London, 1998, 103 S.
Detaillierte Anweisung an Studenten zur Planung, Durchführung und Überwachung von Projekten.

R. W. Sebesta Concepts of Programming Languages
Benjamin/Cummings, Redwood City, 1993, 560 S.

I. Sommerville Software Engineering
Addison-Wesley, Boston, 1992, 688 S.
Wie man ein Programmierprojekt organisiert; Werkzeuge, Methoden; sprachenunabhängig

N. Wirth Systematisches Programmieren
Teubner, Stuttgart, 1993, 160 S.
Allgemeine Einführung ins Programmieren, PASCAL-nahe

12. Programmieren in C/C++/Objective C

- Newsgruppen:
  comp.lang.c
  comp.std.c
  comp.lang.object
  comp.lang.c++
  comp.lang.objective-c
  comp.std.c++
  de.comp.lang.c
  de.comp.lang.c++

G. Booch Object-Oriented Analysis and Design with Applications
Benjamin + Cummings, Redwood City, 1994, 590 S.

U. Breymann Designing Components with the C++ STL
Addison-Wesley, Boston, 2000, 320 S.

B. J. Cox, A. J. Novobilski Object-Oriented Programming
Addison-Wesley, Boston, 1991, 270 S.
Objective C

P. A. Darnell, P. E. Margolis C: A Software Engineering Approach
Springer, Berlin + Heidelberg, 1996, 500 S.

H. M. Deitel, P. J. Deitel C How to Program
Prentice Hall, Englewood Cliffs, 1994, 926 S.
Enthält auch C++. Ausgeprägtes Lehrbuch.

J. Hanly, E. Koffman Problem Solving and Program Design in C
Addison-Wesley, Boston, 1999, 276 S.

J. Hanly, E. Koffman C Program Design for Engineers
Addison-Wesley, Boston, 2001, 679 S.

S. P. Harbison, G. L. Steele C – A Reference Manual
Prentice Hall, Englewood Cliffs, 1995, 470 S.
Vielfach empfohlenes Nachschlagewerk, K+R und ANSI/ISO.

T. Jensen A Tutorial on Pointers and Arrays in C
`http://www.netcom.com/ tjensen/ptr/pointers.htm`

N. M. Josuttis The C++ Standard Library – A Tutorial and Reference
Addison-Wesley, Boston, 1999, 832 S.
`http://www.josuttis.de/libbook/`

B. W. Kernighan, D. M. Ritchie The C Programming Language
Deutsche Übersetzung: Programmieren in C
Zweite Ausgabe, ANSI C
Hanser Verlag, München, 1990, 283 S.
Standardwerk zur Programmiersprache C, Lehrbuch

R. Klatte u. a. C-XSC
Springer, Berlin + Heidelberg, 1993, 269 S.
C++-Klassenbibliothek für wissenschaftliches Rechnen

A. Koenig Accelerated C++: Practical Programming by Example
Addison-Wesley, Boston, 2000, 352 S.

S. Kuhlins, M. Schader Die C++-Standardbibliothek
Springer, Berlin + Heidelberg, 2002, 421 S.
Einführung in die C++ Standard Library einschl. der STL

D. Lewine POSIX Programmer's Guide
O'Reilly, Sebastopol, 1991, 634 S.
Mit Referenz der ANSI-C- und der POSIX-Funktionen

D. Libes Obfuscated C and Other Mysteries
Wiley, New York, 1993, 413 S.

S. Lippman, J. Lajoie C++ Primer
Addison-Wesley, Boston, 3. Aufl. 1998, 1296 S.
Verbreitetes Lehrbuch für Anfänger, enthält auch ANSI-C

N. Matthew, R. Stones Beginning Linux Programming
Wrox Press, Chicago, 1999, 950 S.

N. Matthew, R. Stones Professional Linux Programming
Wrox Press, Chicago, 2000, 1155 S.
Betriebssystemnahe Fragen der Programmierung in C/C++

T. Misfeldt et al. The Elements of C++ Style
Cambridge University Press, 2004, 182 S.
Regeln für gute C++-Programme

S. Oualline Practical C Programming
O'Reilly, Sebastopol, 1997, 451 S.

S. Oualline Practical C++ Programming
O'Reilly, Sebastopol, 1995, 581 S.

P. J. Plauger, J. Brodie Referenzhandbuch Standard C
Vieweg, Braunschweig, 1990, 236 S.

P. J. Plauger The Standard C Library
Prentice-Hall, Englewood Cliffs, 1991, 498 S.
Die Funktionen der C-Standardbibliothek nach ANSI

P. J. Plauger The Draft Standard C++ Library
Prentice-Hall, Englewood Cliffs, 1994, 590 S.
Die Funktionen der C++-Standardbibliothek nach ANSI

R. Robson Using the STL
Springer, Berlin + Heidelberg, 1998, 421 S.

M. Schader, S. Kuhlins Programmieren in C++
Springer, Berlin + Heidelberg, 1998, 386 S.
Lehrbuch und Nachschlagewerk, mit Übungsaufgaben

K. Schmaranz Softwareentwicklung in C
Springer, Berlin + Heidelberg, 2002, 400 S.

K. Schmaranz Softwareentwicklung in C++
Springer, Berlin + Heidelberg, 2003, 570 S.

B. Stroustrup The C++ Programming Language
bzw. Die C++ Programmiersprache
Addison-Wesley, Boston/Bonn, 2000, 1024 S.
Lehrbuch für Fortgeschrittene, der Klassiker für C++

13. Netze allgemein (Internet, OSI)

– Newsgruppen:
comp.infosystems.*
comp.internet.*
comp.protocols.*
alt.best.of.internet
alt.bbs.internet
alt.internet.*
de.comm.internet
de.comp.infosystems

– EFF's Guide to the Internet
`http://www.eff.org/pub/Publications/EFF\_Net\_Guide/`
Einführung in die Dienste des Internet

S. Carl-Mitchell, J. S. Quarterman Practical Internetworking with TCP/IP and UNIX
Addison-Wesley, Boston, 1993, 432 S.

D. E. Comer Internetworking with TCP/IP (4 Bände)
Prentice-Hall, Englewood Cliffs, I. Band 1991, 550 S.
II. Band 1991, 530 S., 88 DM; IIIa. Band (BSD) 1993, 500 S.
IIIb. Band (AT&T) 1994, 510 S.
Prinzipien, Protokolle und Architektur des Internet

H. Hahn, R. Stout The Internet Complete Reference
Osborne MacGraw-Hill, Berkeley, 1994, 818 S.
Das Netz und seine Dienste von Mail bis WWW; Lehrbuch und Nachschlagewerk für Benutzer des Internet

C. Hunt TCP/IP Netzwerk-Administration
O'Reilly, Sebastopol, 1998, 632 S.

B. P. Kehoe Zen and the Art of the Internet
`ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/general/zen.ps.gz`
1992, 100 S., Postscript
Einführung in die Dienste des Internet

O. Kirch, T. Dawson Linux Network Administrator's Guide
O'Reilly, Sebastopol, 2000, 500 S.

E. Krol The Hitchhikers Guide to the Internet
`ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/general/hitchhg.txt`
1987, 16 S., ASCII
Erklärung einiger Begriffe aus dem Internet

E. Krol The Whole Internet
O'Reilly, Sebastopol, 1992, 376 S.

J. F. Kurose, K. W. Ross Computer Networking
Addison-Wesley, Boston, 2003, 784 S.

M. Scheller u. a. Internet: Werkzeuge und Dienste
Springer, Berlin + Heidelberg, 1994, 280 S.
`http://www.ask.uni-karlsruhe.de/books/inetwd.html`

A. S. Tanenbaum Computer Networks
Prentice-Hall, London, 1996, 848 S.
Einführung in Netze mit Schwerpunkt auf dem OSI-Modell

14. Netzdienste Einzelthemen

– Newsgruppen:
comp.theory.info-retrieval
comp.databases.*

P. Albitz, C. Liu DNS and BIND
O'Reilly, Sebastopol, 1998, 482 S.
Internet-Adressen und -Namen, Name-Server

B. Costales, E. Allman sendmail
O'Reilly, Sebastopol, 1997, 1021 S.
Das wichtigste netzseitige Email-Programm (MTA) ausführlich dargestellt, keine leichte Kost, aber unentbehrlich

J. E. Hellbusch Barrierefreies Webdesign
KnowWare, `www.knowware.de/`, 2001, 86 S.
Hinweise zur Gestaltung von Webseiten, kompakt und verständlich

P. J. Lynch, S. Horton Web Style Guide
Yale University Press, New Haven, 1999, 165 S.
Gestaltung und Organisation von Webseiten, wenig Technik

C. Meinel, H. Sack WWW
Springer, Berlin + Heidelberg, 2004, 1200 S.
Internet-Grundlagen, HTTP, HTML, CSS, XML, CGI

S. Münz, W. Nefzger HTML 4.0 Handbuch
Franzis, München, 1999, 992 S.
Deutsches Standardwerk zum Schreiben von Webseiten, abgewandelt auch unter dem Titel *Selfhtml* an mehreren Stellen im Netz verfügbar.

J. Niederst Web Design in a Nutshell
O'Reilly, Sebastopol, 1999, 560 S.
Das gesamte Web zum Nachschlagen, viel Technik

A. Schwartz Managing Mailing Lists
O'Reilly, Sebastopol, 1998, 320 S.
Majordomo, Listserv, List Processor und Smartlist

S. Spainhour, R. Eckstein Webmaster in a Nutshell
O'Reilly, Sebastopol, 1999, 523 S.
HTML, CSS, XML, JavaScript, CGI und Perl, PHP, HTTP, Apache

W. R. Stevens UNIX Network Programming
Vol. 1: Networking APIs: Sockets and XTI
Prentice Hall, Englewood Cliffs, 1998, 1009 S.
Vol. 2: Interprocess Communication
Prentice Hall, Englewood Cliffs, 1999, 592 S.
C-Programme für Clients und Server der Netzdienste

15. Sicherheit

– Newsgruppen:
comp.security.*
comp.virus
sci.crypt
alt.security.*
alt.comp.virus
de.comp.security

- RFC 1244 (FYI 8): Site Security Handbook
  ftp://ftp.nic.de/pub/rfc/rfc1244.txt
  1991, 101 S., ASCII
  Sicherheits-Ratgeber für Internet-Benutzer
- Department of Defense Trusted Computer Systems
  Evaluation Criteria (Orange Book)
  ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/secur/orange-book.gz
  1985, 120 S., ASCII. Abgelöst durch:
  Federal Criteria for Information Technology Security
  ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/secur/fcvol1.ps.gz
  ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/secur/fcvol2.ps.gz
  1992, 2 Bände mit zusammen 500 S., Postscript
  Die amtlichen amerikanischen Sicherheitsvorschriften
- Linux Hacker's Guide
  Markt + Technik, München, 1999, 816 S.

F. L. Bauer Kryptologie
Springer, Berlin + Heidelberg, 1994, 369 S.

R. L. Brand Coping with the Threat of Computer Security Incidents
A Primer from Prevention through Recovery
ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/secur/primer.ps.gz
1990, 44 S., Postscript

D. A. Curry Improving the Security of Your UNIX System
ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/net/secur/secdoc.ps.gz
1990, 50 S., Postscript
Hilfe für UNIX-System-Verwalter, mit Checkliste

S. Garfinkel, G. Spafford Practical Unix + Internet Security
O'Reilly, Sebastopol, 1996, 971 S.
Breit angelegte, verständliche Einführung in Sicherheitsthemen

B. Schneier Angewandte Kryptographie
Addison-Wesley, Bonn, 1996, 844 S.

M. Schumacher, U. Roedig, M.-L. Moschgath Hacker Contest
Springer, Berlin + Heidelberg, 2003, 300 S.

16. Computerrecht

- Newsgruppen:
  comp.society.privacy
  comp.privacy
  comp.patents
  alt.privacy
  de.soc.recht
  de.soc.datenschutz
- World Intellectual Property Organization (WIPO)
  http://www/wipo.int/
- Juristisches Internetprojekt Saarbrücken
  http://www.jura.uni-sb.de/
- Netlaw Library (Universität Münster)
  http://www.jura.uni-muenster.de/netlaw/
- Online-Recht http://www.online-recht.de/
- Computerrecht (Beck-Texte)
  Beck, München, 1994

U. Dammann, S. Simitis Bundesdatenschutzgesetz
Nomos Verlag, Baden-Baden, 1993, 606 S.
BDSG mit Landesdatenschutzgesetzen und Internationalen
Vorschriften; Texte, kein Kommentar

G. v. Gravenreuth Computerrecht von A – Z (Beck Rechtsberater)
Beck, München, 1992

H. Hubmann, M. Rehbinder Urheber- und Verlagsrecht
Beck, München, 1991, 319 S.

A. Junker Computerrecht. Gewerblicher Rechtsschutz,
Mängelhaftung, Arbeitsrecht. Reihe Recht und Praxis
Nomos Verlag, Baden-Baden, 1988, 267 S.

F. Koch Handbuch Software- und Datenbank-Recht
Springer, Berlin + Heidelberg, 2003, 1000 S.

D. Kröger, M. A. Gimmy Handbuch zum Internetrecht
Springer, Berlin + Heidelberg, 2. Auflage 2002, 1000 S.

17. Geschichte der Informatik

- Newsgruppen:
  comp.society.folklore
  alt.folklore.computers
  de.alt.folklore.computer
- Kleine Chronik der IBM Deutschland
  1910 – 1979, Form-Nr. D12-0017, 138 S.
  1980 – 1991, Form-Nr. D12-0046, 82 S.
  Reihe: Über das Unternehmen, IBM Deutschland
- Die Geschichte der maschinellen Datenverarbeitung Band 1
  Reihe: Enzyklopädie der Informationsverarbeitung
  IBM Deutschland, 228 S., Form-Nr. D12-0028
- 100 Jahre Datenverarbeitung Band 2
  Reihe: Über die Informationsverarbeitung
  IBM Deutschland, 262 S., Form-Nr. D12-0040
- Open Source
  O'Reilly, Köln, 1999, 70 S.

P. E. Ceruzzi A History of Modern Computing
MIT Press, Cambridge/USA, 1998, 400 S.
Computergeschichte seit 1945 aus nordamerikanischer Sicht

O. A. W. Dilke Mathematik, Maße und Gewichte in
der Antike (Universalbibliothek Nr. 8687 [2])
Reclam, Stuttgart, 1991, 135 S.

M. Hauben, R. Hauben Netizens – On the History and
Impact of Usenet and the Internet
IEEE Computer Society Press, Los Alamitos, 1997, 345 S.
`www.columbia.edu/~hauben/netbook/`

A. Hodges Alan Turing, Enigma
Kammerer & Unverzagt, Berlin, 1989, 680 S.

D. M. Lehmann Der EDV-Pionier Nikolaus Joachim Lehmann
Dr. Hänsel-Hohenhausen, Frankfurt (M), 2002,

S. Levy Hackers – Heroes of the Computer Revolution
Penguin Books, London, 1994, 455 S.

R. Oberliesen Information, Daten und Signale
Deutsches Museum, rororo Sachbuch Nr. 7709 (vergriffen)

D. Shasha, C. Lazere Out of Their Minds
Springer, Berlin + Heidelberg, 1995, 295 S.
Biografien berühmter Computerpioniere

D. Siefkes u. a. Pioniere der Informatik
Springer, Berlin + Heidelberg, 1998, 160 S.
Interviews mit fünf europäischen Computerpionieren

B. Sterling A short history of the Internet
`ftp://ftp.ciw.uni-karlsruhe.de/pub/docs/history/origins`
1993, 6 S., ASCII

K. Zuse Der Computer - Mein Lebenswerk
Springer, Berlin + Heidelberg, 3. Aufl. 1993, 220 S.
Autobiografie Konrad Zuses

18. Allgemeinwissen und Philosophie

– Newsgruppen:
comp.ai.philosophy
sci.philosophy.tech
alt.fan.hofstadter

E. Dyson Release 2.1 – A Design for Living in the Digital Age
Petersen, Hamburg, 2000, 370 S.

D. R. Hofstadter Gödel, Escher, Bach - ein Endloses Geflochtenes Band
dtv/Klett-Cotta, München, 1992, 844 S.

J. Ladd Computer, Informationen und Verantwortung
in: Wissenschaft und Ethik, herausgegeben von H. Lenk
Reclam-Band 8698, Ph. Reclam, Stuttgart

H. Lenk Chancen und Probleme der Mikroelektronik, und:
Können Informationssysteme moralisch verantwortlich sein?
in: Hans Lenk, Macht und Machbarkeit der Technik
Reclam-Band 8989, Ph. Reclam, Stuttgart, 1994, 152 S.

P. Schefe u. a. Informatik und Philosophie
BI Wissenschaftsverlag, Mannheim, 1993, 326 S.
18 Aufsätze verschiedener Themen und Meinungen

K. Steinbuch Die desinformierte Gesellschaft
Busse + Seewald, Herford, 1989, 269 S. (vergriffen)

J. Weizenbaum Die Macht der Computer und die Ohnmacht der Vernunft (Computer Power and Human Reason. From Judgement to Calculation)
Suhrkamp Taschenbuch Wissenschaft 274, Frankfurt (Main), 1990, 369 S.

H. Zemanek Das geistige Umfeld der Informationstechnik
Springer, Berlin + Heidelberg, 1992, 303 S.
Zehn Vorlesungen über Technik, Geschichte und Philosophie des Computers, von einem der Pioniere

19. Zeitschriften

– c't
Verlag Heinz Heise, Hannover, vierzehntägig,
für alle Fragen der Computerei, technisch.
`http://www.ix.de/`

– IX
Verlag Heinz Heise, Hannover, monatlich,
für Anwender von Multi-User-Systemen, technisch.
`http://www.ix.de/`

- The C/C++ Users Journal
  Miller Freeman Inc., USA, monatlich,
  http://www.cuj.com/
- Dr. Dobb's Journal
  Miller Freeman Inc., USA, monatlich,
  http://www.ddj.com/
  Software Tools for the Professional Programmer; viel C und C++

Und noch einige Verlage:

- Addison-Wesley, Bonn,
  http://www.addison-wesley.de/
- Addison Wesley Longman, USA,
  http://www.awl.com/
- Computer- und Literaturverlag, Vaterstetten,
  http://www.cul.de/
- Carl Hanser Verlag, München,
  http://www.hanser.de/
- Verlag Heinz Heise, Hannover,
  http://www.heise.de/
- International Thomson Publishing, Stamford,
  http://www.thomson.com/
- Klett-Verlag, Stuttgart,
  http://www.klett.de/
- MITP-Verlag, Bonn,
  http://www.mitp.de/
- R. Oldenbourg Verlag, München,
  http://www.oldenbourg.de/
- O'Reilly, Deutschland,
  http://www.ora.de/
- O'Reilly, Frankreich,
  http://www.editions-oreilly.fr/
- O'Reilly, USA,
  http://www.ora.com/
- Osborne McGraw-Hill, USA,
  http://www.osborne.com/
- Prentice-Hall, USA,
  http://www.prenhall.com/
- Sams Publishing (Macmillan Computer Publishing), USA,
  http://www.mcp.com/
- Springer-Verlag, Berlin, Heidelberg, New York usw.,
  http://www.springer.de/
- Wrox Press, Chicago, Birmingham, Paris,
  http://www.wrox.com/

Und über allem, mein Sohn, laß dich warnen;
denn des vielen Büchermachens ist kein Ende,
und viel Studieren macht den Leib müde.

Prediger 12, 12

Personenverzeichnis

Sachverzeichnis